Chapter 7: Run-Time Environment-Memory Organization during program execution

1. Introduction

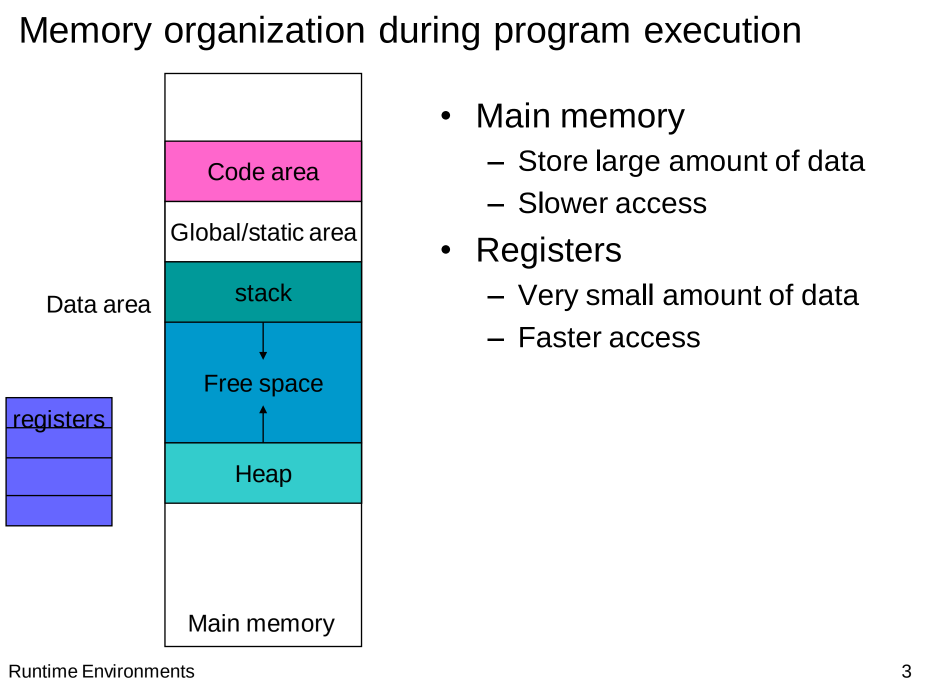
- **Runtime Environment**

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**

- **Almost all programming languages use one of three kinds of runtime environments**

(1) **Fully static** environment; FORTRAN77
(2) **Stack-Based** environment; C C++
(3) **Fully dynamic** environment; LISP

2. Memory Organization during program



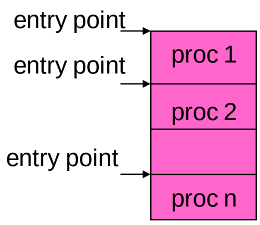
- The memory of a typical computer is divided into:
  - A register area;
  - Addressable Random access memory (RAM):
- The RAM may be divided into:
  A code area;
  A data area.

2.1 Code Area

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

Code Area

- Addresses in code area are static (i.e. no change during execution) for most programming language.
- Addresses are known at compile time.



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

2.2 Data Area

- The same cannot be said for the allocation of data.
- There is one class of data that can be fixed in memory prior to execution that comprises

Data Area

- Addresses in data area are static for some data and dynamic for others.
  - Static data are located in static area.
  - Dynamic data are located in stack or heap.
    - Stack (LIFO allocation) for procedure activation record, etc.
    - Heap for user allocated memory, etc.

- 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**
  - Const 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.

2.3 Registers

Registers

- General-purpose registers
  - Used for calculation
- Special purpose registers
  - Program counter (pc)
  - Stack pointer (sp)
  - Frame pointer (fp)
  - Argument pointer (ap)

2.1 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.
In some organization the stack and heap are allocated separate sections of memory.

2.2 Procedure activation record (An important unit of memory allocation)

Memory allocated for the local data of a procedure or function.

An activation record must contain 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.

- 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.

Calling Sequence

- Sequence of operations that must be done for procedure calls
  - Call sequence
    - Sequence of operations performed during procedure calls
      - Find the arguments and pass them to the callee.
      - Save the caller environment, i.e. local variables in activation records, return address.
      - Create the callee environment, i.e. local variables in activation records, callee's entry point.
  - Return sequence
    - Sequence of operations performed when return from procedure calls
      - Find the arguments and pass them back to the caller.
      - Free the callee environment.
      - Restore the caller environment, including PC.

Issues in Call Sequence

- Which part of the call sequence is included in the caller code? Which is in the callee code?
  - Save space in the code segment if the call sequence is included in the callee code.
  - Normally, the caller finds the arguments and provides them to the callee.
- Which operation is supported in hardware?
  - The more operations supported in hardware, the lower cost (i.e. execution time and space for code) is.

Runtime Environments 8

- **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***
  - 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

Chapter 8: **Code Generation**

1. Introduction

- Purpose: Generate executable code for a target machine that is a faithful representation of the semantics of the source code
- Depends not only on the characteristics of ***the source language*** but also on detailed information about ***the target architecture***, the structure of the ***runtime environment***, and t***he operating system*** running on the target machine

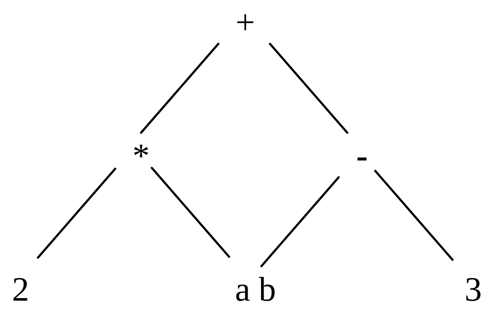
2. **Intermediate Code and Data Structure for code Generation**

2.1 Three-Address Code

- A data structure that represents the source program during translation is called an **intermediate representation, or IR**, for short
- Such an intermediate representation that resembles target code is called intermediate code
  - Intermediate code is particularly useful when the goal of the compiler is **to produce extremely efficient code**;
  - Intermediate code can also be useful in making **a compiler more easily retarget-able**.
- Study two popular forms of intermediate code: **Three -Address code** and **P-code**
- The most basic instruction of three-address code is designed to represent the evaluation of arithmetic expressions and has the following general form:

X=y op z

2*a+(b-3) with syntax tree



The corresponding three-address code is

T1 = 2 * a

T2 = b – 3

T3 = t1 + t2

Sample TINY program:

```
  { sample program
    in TINY language -- computes factorial
  }
read x ;  { input an integer }
if 0 < x then { don't compute if x <= 0 }
   fact:=1;
   repeat
```

```
      fact:=fact*x;
    x:=x-1
until x=0;
write fact { output factorial of x }
    ends
```

- The Three-address codes for above TINY program

```
            read x
    t1=x>0
            if_false t1 goto L1
            fact=1
            label L2
            t2=fact*x
            fact=t2
            t3=x-1
            x=t3
            t4= x= =0
            if_false t4 goto L2
            write fact
            label L1
            halt
```

2.2 **Data Structures for the Implementation of Three-Address Code**

- The most common implementation is to implement three-address code as quadruple, which means that four fields are necessary:
  - **One for the operation and three for the addresses**
- A different implementation of three-address code is called a triple:
  - **Use the instructions themselves to represent the temporaries.**
- It requires that each three-address instruction be reference-able, either as an index in an array or as a pointer in a linked list.
- Quadruple implementation for the three-address code of the previous example

```
(rd, x , _ , _ )
(gt, x, 0, t1 )
(if_f, t1, L1, _ )
(asn, 1,fact, _ )
(lab, L2, _ , _ )
(mul, fact, x, t2 )
(asn, t2, fact, _ )
(sub, x, 1, t3 )
(asn, t3, x, _ )
(eq, x, 0, t4 )
(if_f, t4, L2, _)
(wri, fact, _, _ )
(lab, L1, _ , _ )
(halt, _, _, _ )
```

- **A representation of the three-address code of the previous example as triples**

```
(0)      (rd, x , _  )
(1)      (gt, x, 0)
(2)      (if_f, (1), (11) )
```

(3) (asn, 1,fact)
(4) (mul, fact, x)
(5) (asn, (4), fact)
(6) (sub, x, 1)
(7) (asn, (6), x)
(8) (eq, x, 0)
(9) (if_f, (8), (4))
(10) (wri, fact, _)
(11) (halt, _, _)