A Book Chapter

By Narendra Kumar Chahar

on **Local and Shared Data, Parameters and Parameter Transmission**

Index:

Chapter 9

Local and Shared Data, Parameters and Parameter Transmission

A. Parameter transmission

Subprograms need mechanisms to exchange data.

Arguments - data objects sent to a subprogram to be processed Obtained through

- parameters
- non-local references

Results - data object or values delivered by the subprogram Returned through

- parameters
- assignments to non-local variables
- explicit function values

6. Actual and Formal Parameters

A formal parameter is a particular kind of local data object within a subprogram.
It has a name, the declaration specifies its attributes.

An actual parameter is a data object that is shared with the caller subprogram.
Might be:

- a local data object belonging to the caller,
- a formal parameter of the caller,
- a nonlocal data object visible to the caller,
- a result returned by a function invoked by the caller and immediately transmitted to the called subprogram.

Establishing a Correspondence

Positional correspondence– pairing actual and formal parameters based on their respective positions in the actual- and formal- parameter lists.

Correspondence by explicit name– the name is paired explicitly by the caller.

7. **Methods for transmitting parameters**

Call by name– the actual parameter is substituted in the subprogram.

Call by reference– a pointer to the location of the data object is made available to the subprogram. The data object does not change position in memory.

Call by value– the value of the actual parameter is copied in the location of the formal parameter.

Call by value-result– same as call by value, however at the end of execution the result is copied into the actual parameter.

Call by constant value– if a parameter is transmitted by constant value, then no change in the value of the formal parameter is allowed during program execution.

Call by result– a parameter transmitted by result is used only to transmit a result back from a subprogram. The initial value of the actual-parameter data object makes no difference and cannot be used by the subprogram.

Note: Often "pass by" is used instead of "call by" .

Examples:

Pass by name in Algol	**Pass by reference in FORTRAN**
Procedure S (el, k);	SUBROUTINE S (EL, K)
integer el, k;	K = 2
begin	EL = 0
k:=2; el := 0	RETURN
end;	END
A[1] := A[2] := 1;	A(1) = A(2) = 1
i := 1;	I = 1
S(A[i],i);	CALL S (A(I), I)

Pass by name:

After calling S(A[i],i), the effect is as if the procedure were

i := 2;

A[i] := 0;

As a result A[2] becomes 0.

On exit we have

i = 2, A[1] = 1, A[2] = 0.

Pass by reference:

Since at the time of call **i** is 1, the formal parameter **el** is linked to the address of A(1).

Thus it is A(1) that becomes 0.

On exit we have: i = 2, A(1) = 0, A(2) = 1

8. **Transmission semantics**

Types of parameters:

- input information
- output information (the result)
- both input and output

The three types can be accomplished by copying or using pass-by-reference

Return results:

- Using parameters
- Using functions with a return value

9. **Implementation of parameter transmission**

Implementing formal parameters:

Storage - in the activation record
Type: Local data object of type T in case of pass by value, pass by value-result, pass by result
Local data object of type pointer to T in case of pass by reference

Call by name implementation: the formal parameters are subprograms that evaluate the actual parameters.

Actions for parameter transmission:

- associated with the point of call of the subprogram

  each actual parameter is evaluated in the referencing environment of the calling program, and list of pointers is set up.

- associated with the entry and exit in the subprogram

  on entry:
  copying the entire contents of the actual parameter in the formal parameter, or copying the pointer to the actual parameter
  on exit:
  copying result values into actual parameters or copying function values into registers

These actions are performed by prologue and epilogue code generated by the compiler and stored in the segment code part of the activation record of the subprogram.

Thus the compiler has two main tasks in the implementation of parameter transmission

- It must generate the correct executable code for transmission of parameters, return of results, and each reference to a formal-parameter name.
- It must perform the necessary static type checking to ensure that the type of each actual-parameter data object matches that declared for the corresponding formal parameter

2. **Explicit common environment**

This method of sharing data objects is straightforward.

Specification: A common environment that is similar to a local environment, however it is not a part of any single subprogram.

It may contain: definitions of variables, constants, types.
It cannot contain: subprograms, formal parameters.

Implementation:as a separate block of memory storage.

Special keywords are used to specify variables to be shared.

Shared Data Types:

The architectural framework must operate on pretty large grains in order to mitigate language primitive overheads and to exploit a significant efficiency in communications. Moreover, we must ensure that SDT typical access patterns exploit a good spatial locality.

As far as shared regions concern they are simply allocated in segments tailored for their size. The implementation of spread arrays has been extensively studied and tested in. We adopt the very simple strategy to divide them in regular segments. Such segments are spread across the architecture accordingly with a hash function12. Trees are more interesting from out viewpoint. We have already discussed how to declare and populate a spread tree in Chapter 5. In summary in order to use a spread tree the programmer should follow these steps:

1. Instance a spread tree SDT with a C type representing node held data. The operation is performed by means of the e declare tree.
2. Declare a spread tree shared variable using as type the name given to the SDT at the previous step (either statically or dynamically via standard malloc). This phase declares an empty tree.
3. Populate the tree starting from the root using the e add node primitive.

Let us describe now what really happens in the run-time support:

1. Two new types are created. One of them represent the tree, the other its generic node. Both of them are really are C struct. The former type is named according to the SDT requested name. It hold few information such as the number of children and a dummy variable holding a prototype of the generic

node. The latter struct holds the C type used as parameter and some additional information such as an array of references to children.

2. Nothing happens apart for the allocation of the first struct mentioned above.
3. Starting from the reference to the father the run-time decides if the requested child must be placed in the same segment or in a new one. This decision is made according to the mapping policy. Clearly if the father is null a new segment is created (and the node is the root of the tree).

- In the case the child is placed in the father's segment it inherits father segment id. The run-time choose only node displacement within the segment according to the "internal" mapping policy (heap, first-fit). Some information are kept within the segment to trace segment current status.
- In the case the child is placed in a new segment a segment id is requested and a suitable memory room is allocated (via malloc). Then the previous step is performed to figure out the displacement.

In both cases a reference is composed by using segment id and displacement then returned.

Semaphores

• Dijkstra - 1965

• A semaphore is a data structure consisting of a counter and a queue for storing task descriptors.

• Semaphores can be used to implement guards on the code that accesses shared data structures.

• Semaphores have only two operations, wait and release (originally called P and V by Dijkstra).

• Semaphores can be used to provide both competition and cooperation Synchronization.

Cooperation Synchronization with Semaphores

• Example: A shared buffer

• The buffer is implemented as an ADT with the operations DEPOSIT and FETCH as the only ways to access the buffer.

• Use two semaphores for cooperation: empty spots and full spots.

• The semaphore counters are used to store the numbers of empty spots and full spots in the buffer.

• DEPOSIT must first check empty spots to see if there is room in the buffer.

• If there is room, the counter of empty spots is decremented and the value is inserted.

• If there is no room, the caller is stored in the queue of empty spots.

• When DEPOSIT is finished, it must increment the counter of full spots.

• FETCH must first check full spots to see if there is a value.

– If there is a full spot, the counter of fulls pots is decremented and the value is removed.

– If there are no values in the buffer, the caller must be placed in the queue of full spots.

– When FETCH is finished, it increments the counter of empty spots.

• The operations of FETCH and DEPOSIT on the semaphores are accomplished through two semaphore operations named wait and release.

Semaphores: Wait Operation

wait(a Semaphore)
if a Semaphore‘s counter > 0 then
decrement a Semaphore‘s counter
else
put the caller in a Semaphore‘s queue
attempt to transfer control to a ready task
-- if the task ready queue is empty,
-- deadlock occurs
end

Semaphores: Release Operation

release(a Semaphore)
if a Semaphore‘s queue is empty then increment a Semaphore‘s counter

else

put the calling task in the task ready queue transfer control to a task from a Semaphore‘s queue

end

Producer Consumer Code

```
semaphore fullspots, emptyspots;
fullstops.count = 0;
emptyspots.count = BUFLEN;
task producer;
loop
-- produce VALUE –-
wait (emptyspots); {wait for space}
DEPOSIT(VALUE);
release(fullspots); {increase filled}
end loop;
        end producer;
```

References:

[1]. Domke, J. (2011, June). Parameter learning with truncated message-passing. In *CVPR 2011* (pp. 2937-2943). IEEE.

[2]. Gleeson, J. P., & Porter, M. A. (2018). Message-passing methods for complex contagions. In *Complex Spreading Phenomena in Social Systems* (pp. 81-95). Springer, Cham.

[3]. Accattoli, B., & Guerrieri, G. (2016, November). Open call-by-value. In *Asian Symposium on Programming Languages and Systems* (pp. 206-226). Springer, Cham.

[4]. Šinkarovs, A., Scholz, S. B., Stewart, R., & Vießmann, H. N. (2017, August). Recursive Array Comprehensions in a Call-by-Value Language. In *Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages* (pp. 1-12).

[5]. Ray, B., Posnett, D., Filkov, V., & Devanbu, P. (2014, November). A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering* (pp. 155-165).

[6]. Hansen, P. B. (Ed.). (2013). *The origin of concurrent programming: from semaphores to remote procedure calls*. Springer Science & Business Media.

[7]. Erciyes, K. (2019). Real-Time Programming Languages. In *Distributed Real-Time*

Systems (pp. 251-275). Springer, Cham.