Chapter 8: Procedures/Subprograms

CMSC 124, 1st Semester, AY 2009-10

Kinds

1. Simple call return
2. Recursive procedures
3. Coroutines
4. Exception handlers
5. Scheduled subprograms
6. Tasks or concurrent procedures

Simple Call Return (and the Copy Rule)

- Jumps to a procedure then resumes at the point where the call is made.
- The simple call-return structure can be explained by the copy rule.
- The copy rule is completely observed by FORTRAN and COBOL.

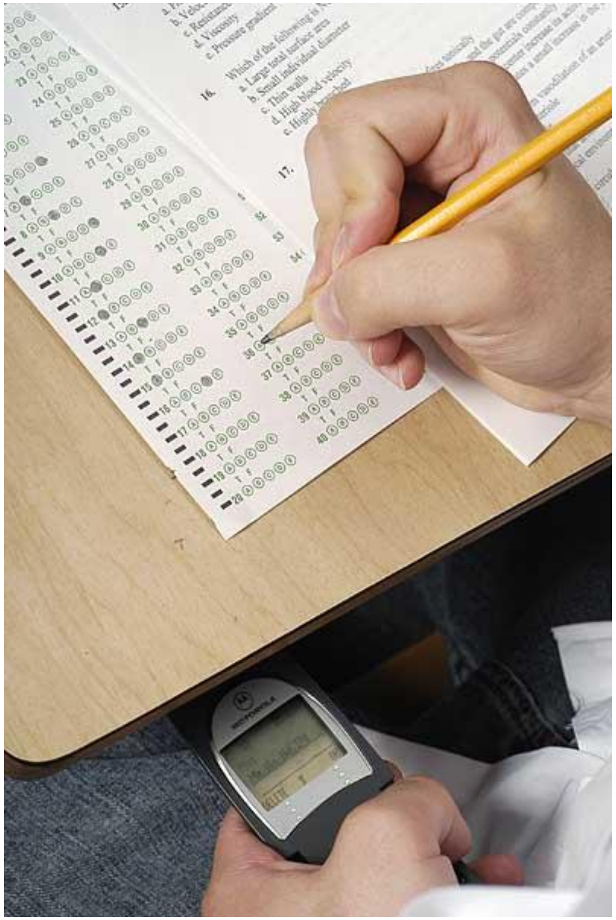
THE COPY RULE

"The effect of the subprogram call statement is the same as would be obtained if the call statement were replaced by a copy of the body of the subprogram (with suitable substitutions for the parameters and conflicting identifiers) before execution."

Chapter 8: Procedures/Subprograms

Limitations of the Copy Rule

- The procedures cannot be recursive.
- Explicit call statements are required.
- Procedures must execute completely at each call.
- Immediate transfer of control at point of call.
- Single execution sequence.



Implementation of Simple Call-Return

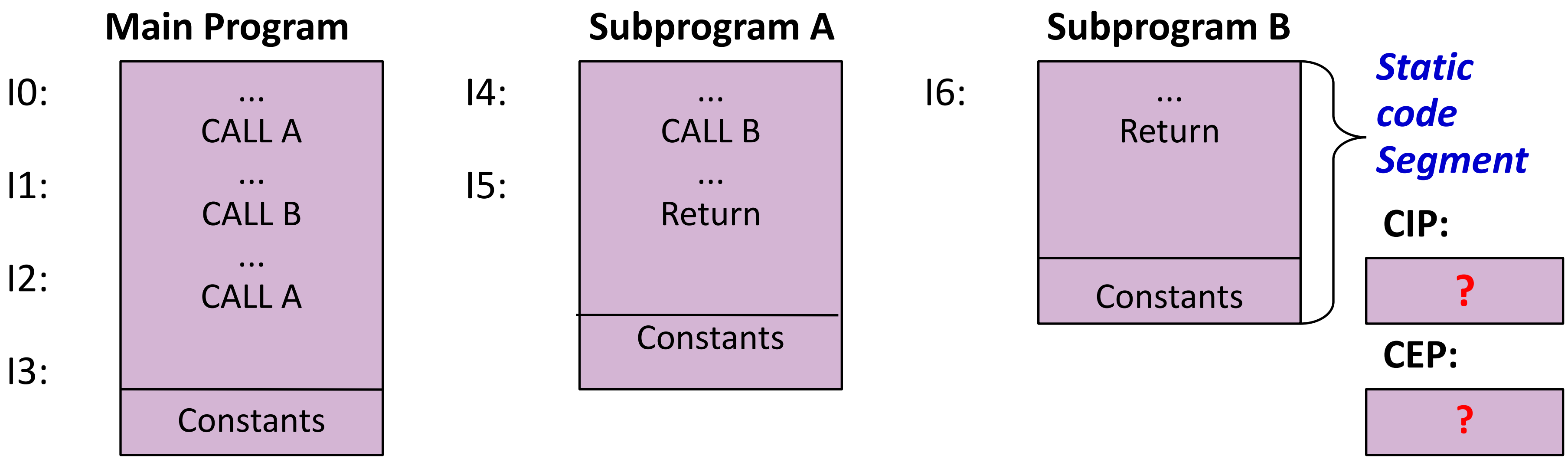
- Simple call-return can be implemented without observing the copy rule.

- An activated procedure has memory allocated to it:
  - ✓ **Code segment** which contains the code itself and the constants. This is invariant.
  - ✓ **Activation record** which houses the local data, instruction and environment pointers. This is destroyed upon return.

- The following has to be tracked, too:
  - **Current instruction pointer (CIP)** –> instruction to be executed.
  - **Current environment pointer (CEP)** –> current activation record that is accessible to the program in execution.

Implementation of Simple Call-Return: Example

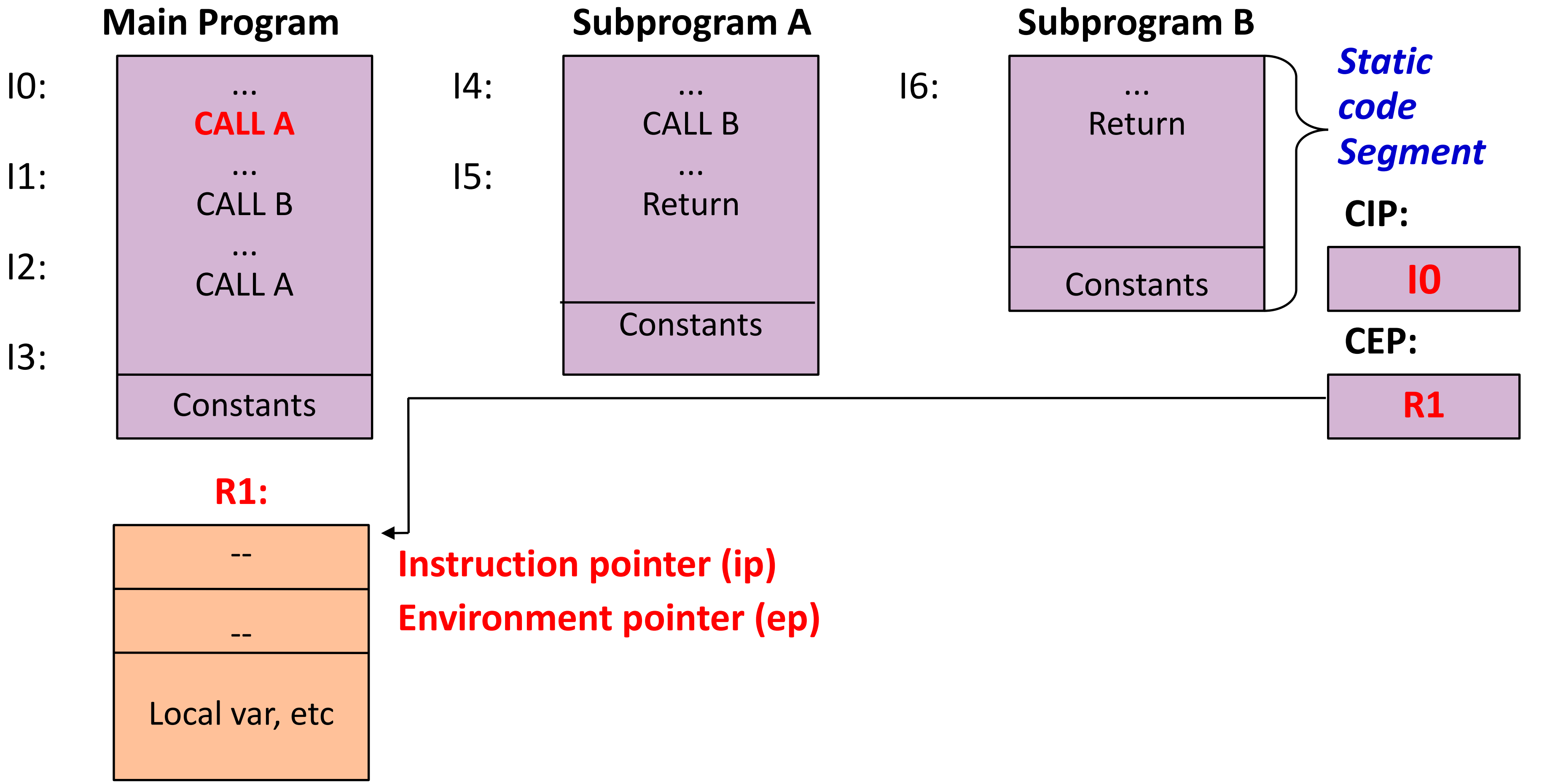
Initial State



Chapter 8: Procedures/Subprograms

Implementation of Simple Call-Return: Example

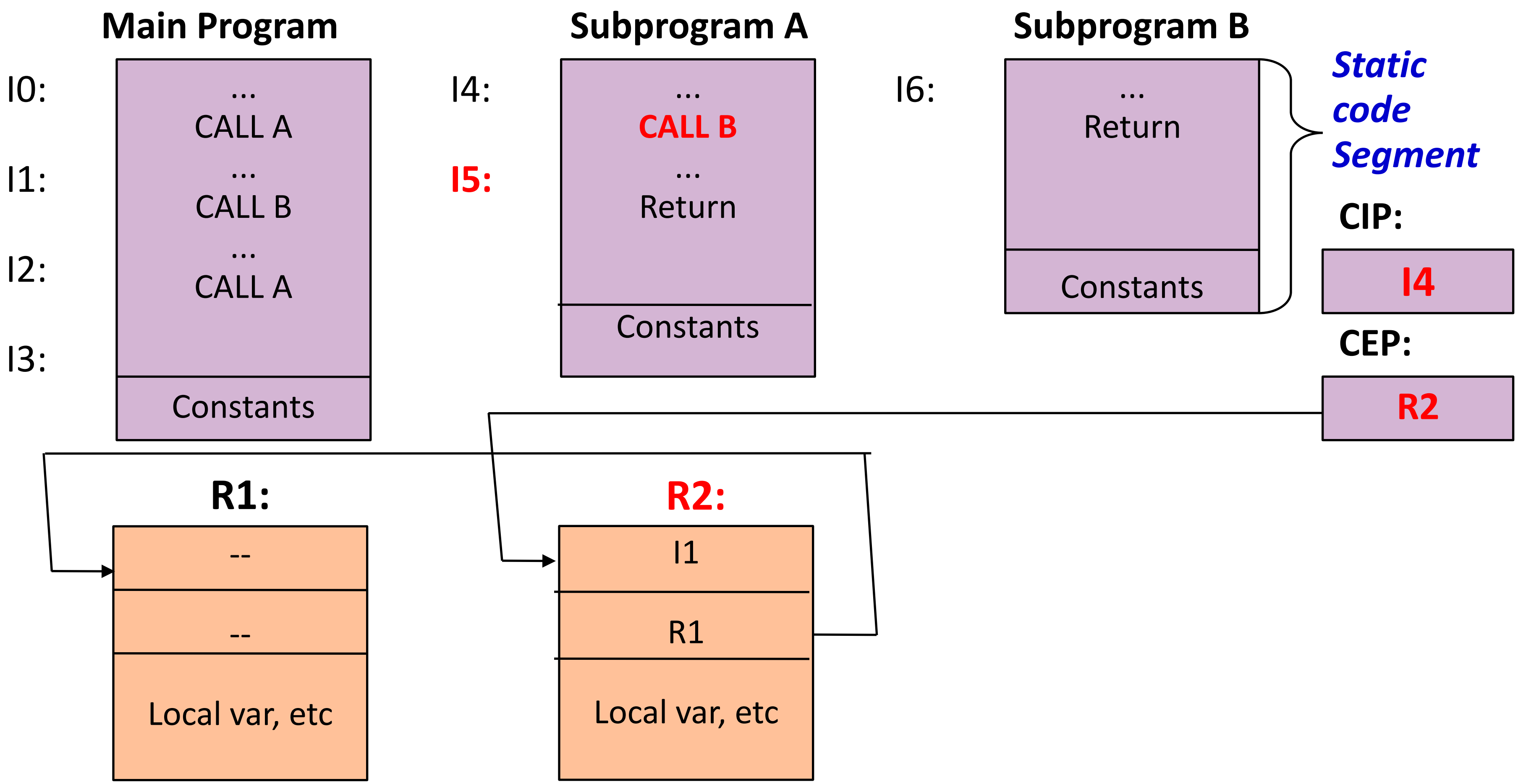
Execution State: Start of execution of main program



Chapter 8: Procedures/Subprograms

Implementation of Simple Call-Return: Example

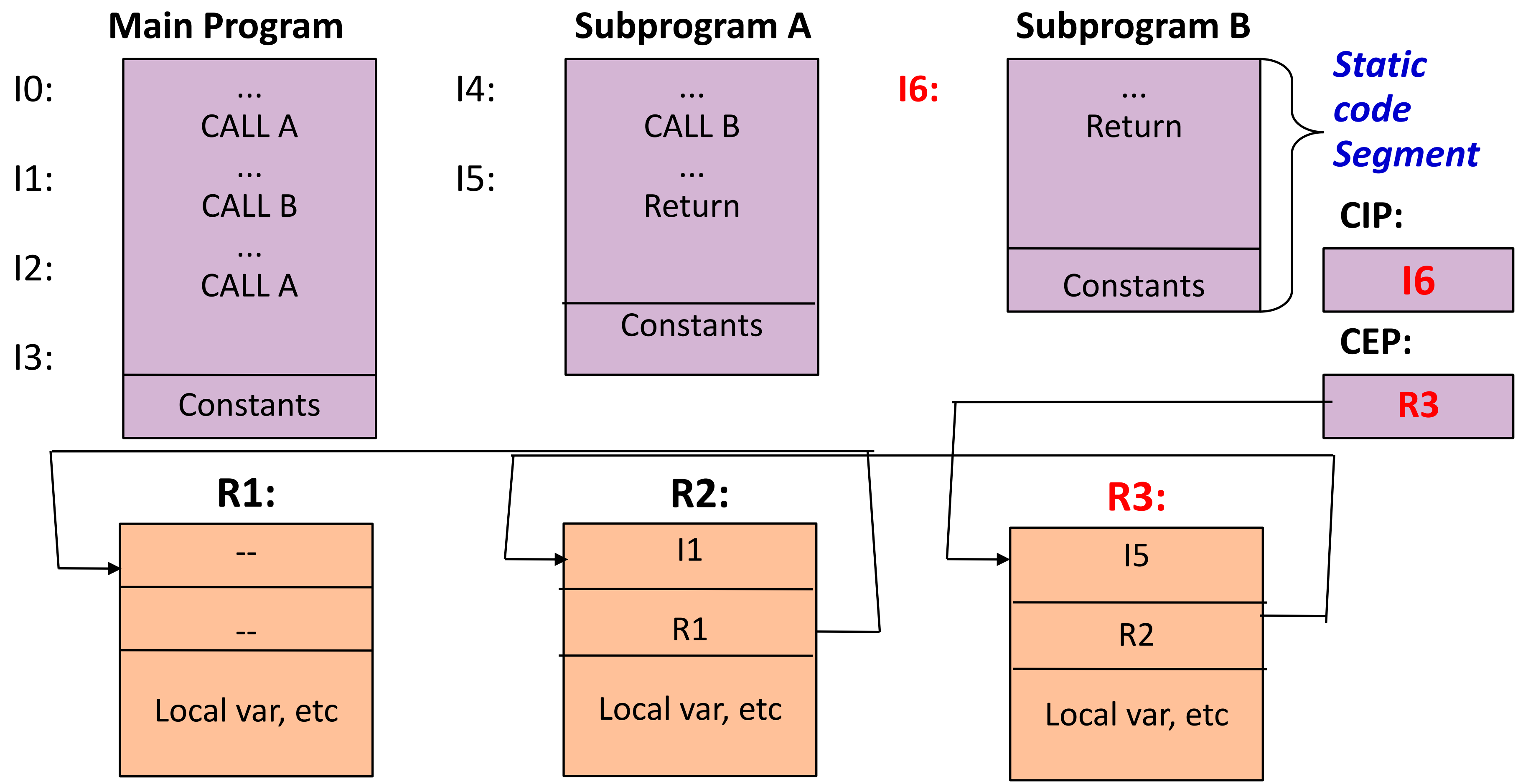
Execution State: Start of execution of subprogram A



Chapter 8: Procedures/Subprograms

Implementation of Simple Call-Return: Example

Execution State: Start of execution of subprogram A



Kinds of Recursive Procedures

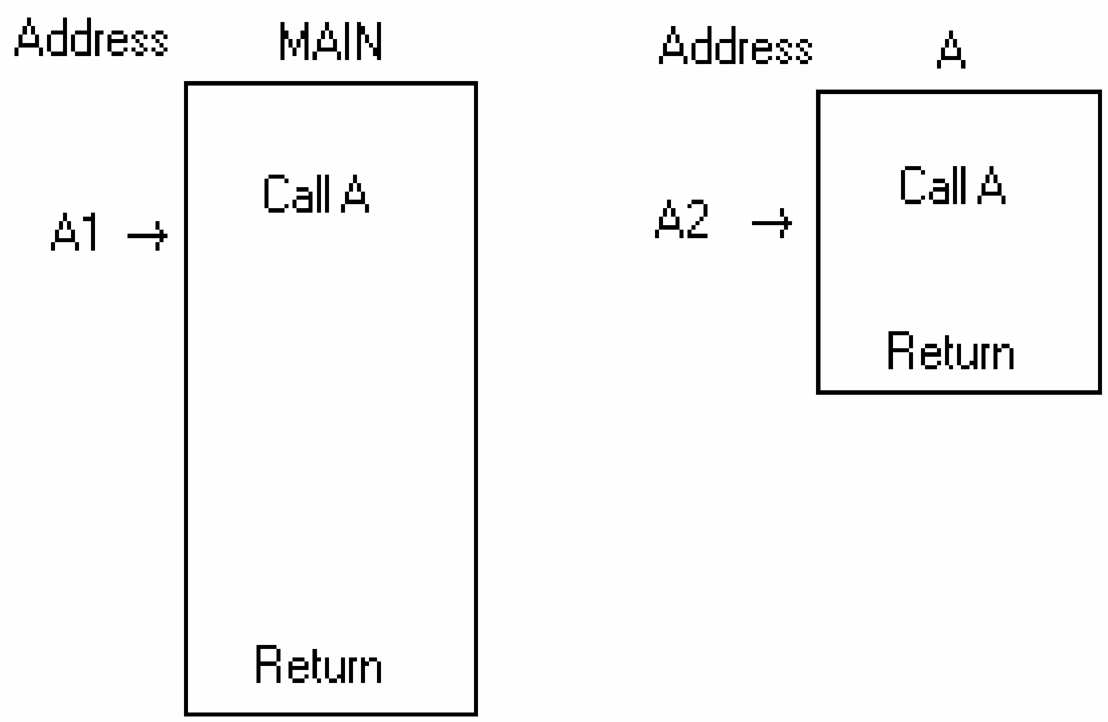
1. **Directly recursive**
   - Contains a call to itself .

2. **Indirectly recursive**
   - Calls another procedure that calls the original procedure or;
   - Initiates a chain of call that eventually call the original procedure.

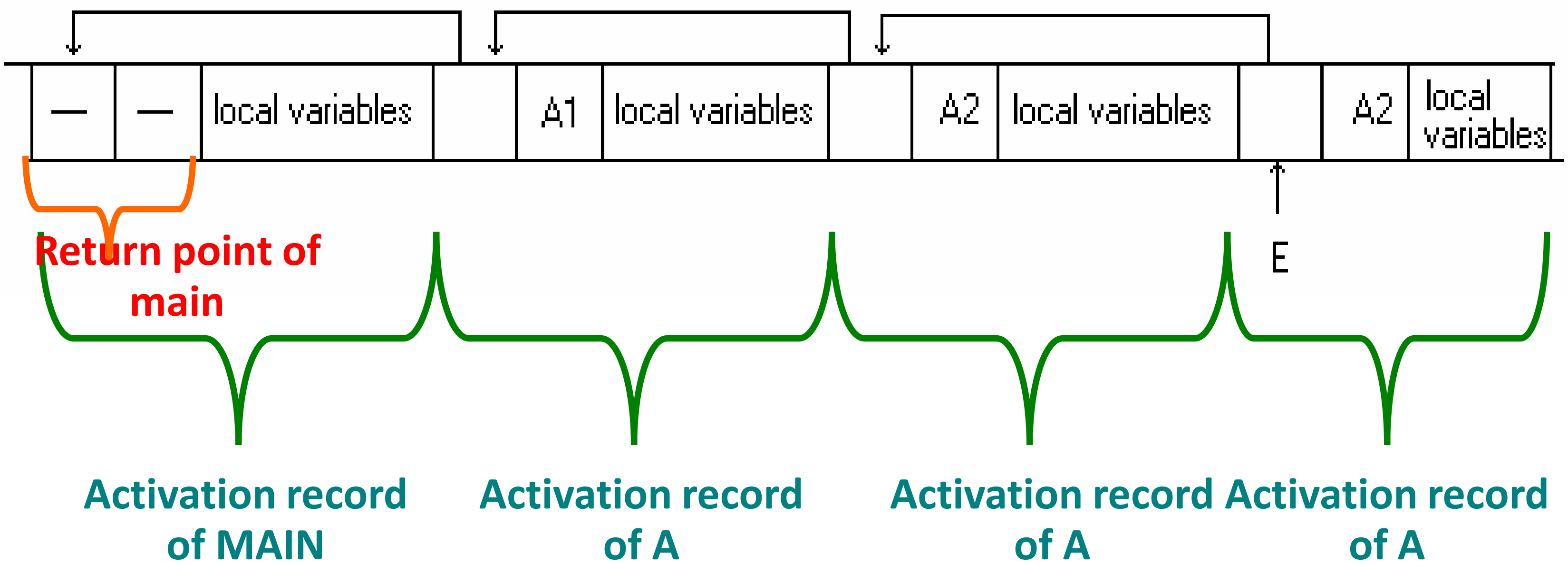
Recursive call creates a second activation of the procedure during the lifetime of the first activation.

Implementation of Recursive Procedures

Code Segment



Activation Record

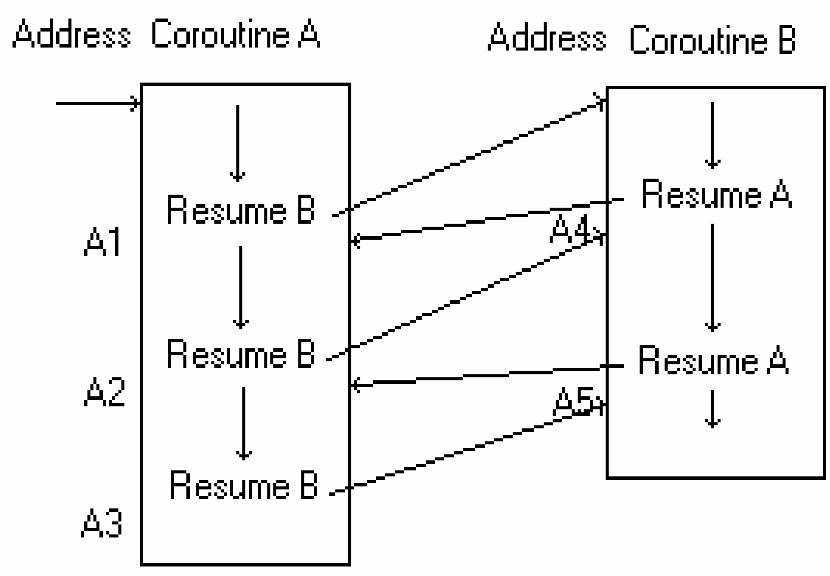


Chapter 8: Procedures/Subprograms

Coroutines

- A procedure that may transfer control to another procedure even though the whole procedure is not completely executed yet.
- The next time control is returned back to the procedure, the execution continues from the first unexecuted instruction.
- Transfer of control to another procedure is achieved by issuing a statement

```
resume coroutinename;
```



Coroutines

- In MODULA-2, a coroutine is also called a **process**.
A new process is created by a call to the procedure:

```
PROCEDURE NEWPROCESS(P; PROC; A: ADDRESS; n:
CARDINAL; VAR p1: ADDRESS);
```

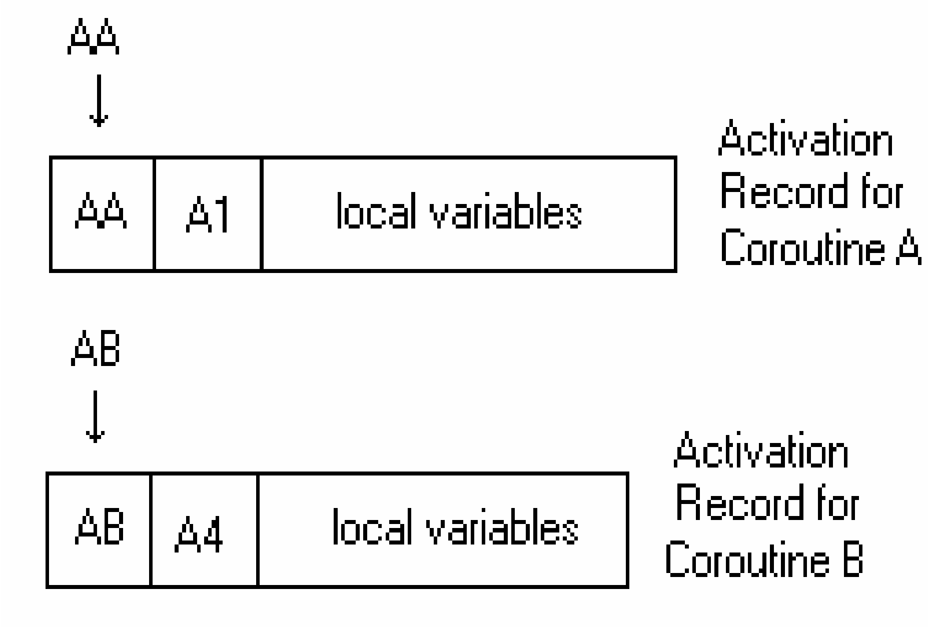
- A transfer of control between two processes is specified by a call to:

```
PROCEDURE TRANSFER(VAR p1, p2: ADDRESS) ;
```

Chapter 8: Procedures/Subprograms

Implementation of Coroutines

- **Activation records** for coroutines must be statically allocated.



Exception Handlers

Exception

• Any unusual event, erroneous or not, that is detectable either hardware or software and that may require special processing.

Exception Handling

• Special processing required after detecting an exception.

Chapter 8: Procedures/Subprograms

Exception Handlers

- **NOT** explicitly-called procedures, which are called when some events or conditions considered **exceptional** becomes true.

- **Eg:**
  - ✓ Array index out of bounds
  - ✓ Division by zero
  - ✓ fopen(filename, "r") is null

Exception Handlers

- **Eg (In Java):**

```
try {
        out.write(b);
        // throws IOException
} catch (IOException e) {
        System.out.println("Output Error");
} finally {
        out.close();
    // ensures that out.close is always executed
}
```

Scheduled Procedures

Usually supported by simulation languages, like:

SIMSCRIPT

• English-like general-purpose simulation language conceived by Harry Markowitz and Bernard Hausner at the RAND Corporation in 1963.

SIMULA

• More at http://en.wikipedia.org/wiki/Simula .

GPSS (General Purpose Simulation System)

• Originally called Gordon's Programmable Simulation System, named after its creator, Geoffrey Gordon.

Scheduled Procedures: Possible Scheduling

1. Before or after another procedure.

```
CALL B AFTER A or CALL A BEFORE B
```

2. When certain conditions are true.

```
CALL A WHEN X > Y
```

3. On the basis of a simulated time scale.

```
CALL A AT TIME = 10
```

4. According to priority designation.

```
CALL A WITH PRIORITY 1
```

Scheduled Procedures

In SIMULA

- activate P
- activate P after O
- activate P before Q
- activate P delay 10.0
- activate P at 10.0

Reactivation in SIMULA

- reactivate P
- reactivate P after O
- reactivate P before Q
- reactivate P delay 10.0
- reactivate P at 10.0

Chapter 8: Procedures/Subprograms

Implementation of Scheduled Procedures

- Scheduled procedure is a **generalized** coroutine.
- In the main function, a system-defined scheduler maintains a list of currently scheduled program activations.

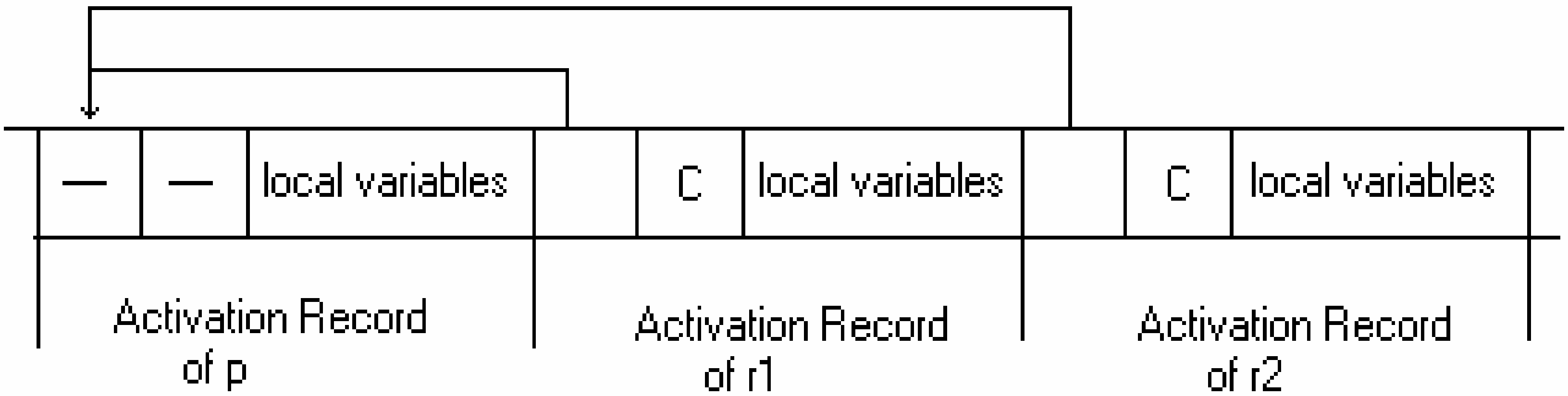
Tasks and Concurrent Procedures

- Procedure that can be executed concurrently with another procedure.

```
program p;
 procedure q;
  begin
  end;
 procedure r1;
  begin
  end;
procedure r2;
 begin
 end;
begin
 cobegin
   r1; r2;
 coend
end.
```

Implementation of Tasks and Concurrent Procedures

- Activation records for concurrent processes.



Parameter Transmission

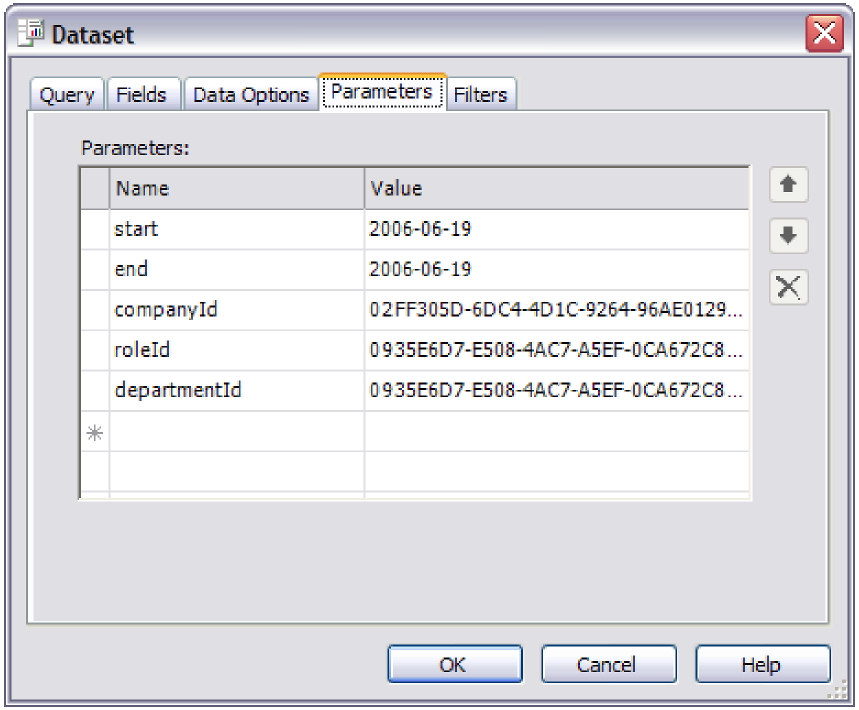
- Data made available from one procedure to another.

Formal Parameter

- Parameter specified in the procedure declaration.

Actual Parameter

- Parameter specified in the procedure call.



Classes of Parameters

Input Parameters

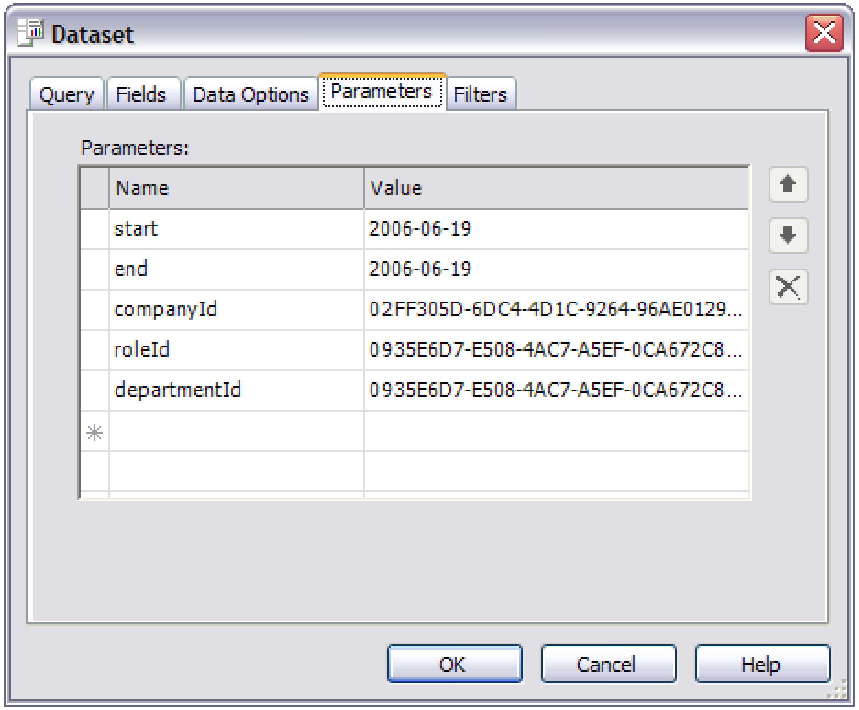
- Supply values from the caller to the called procedure.

Output Parameters

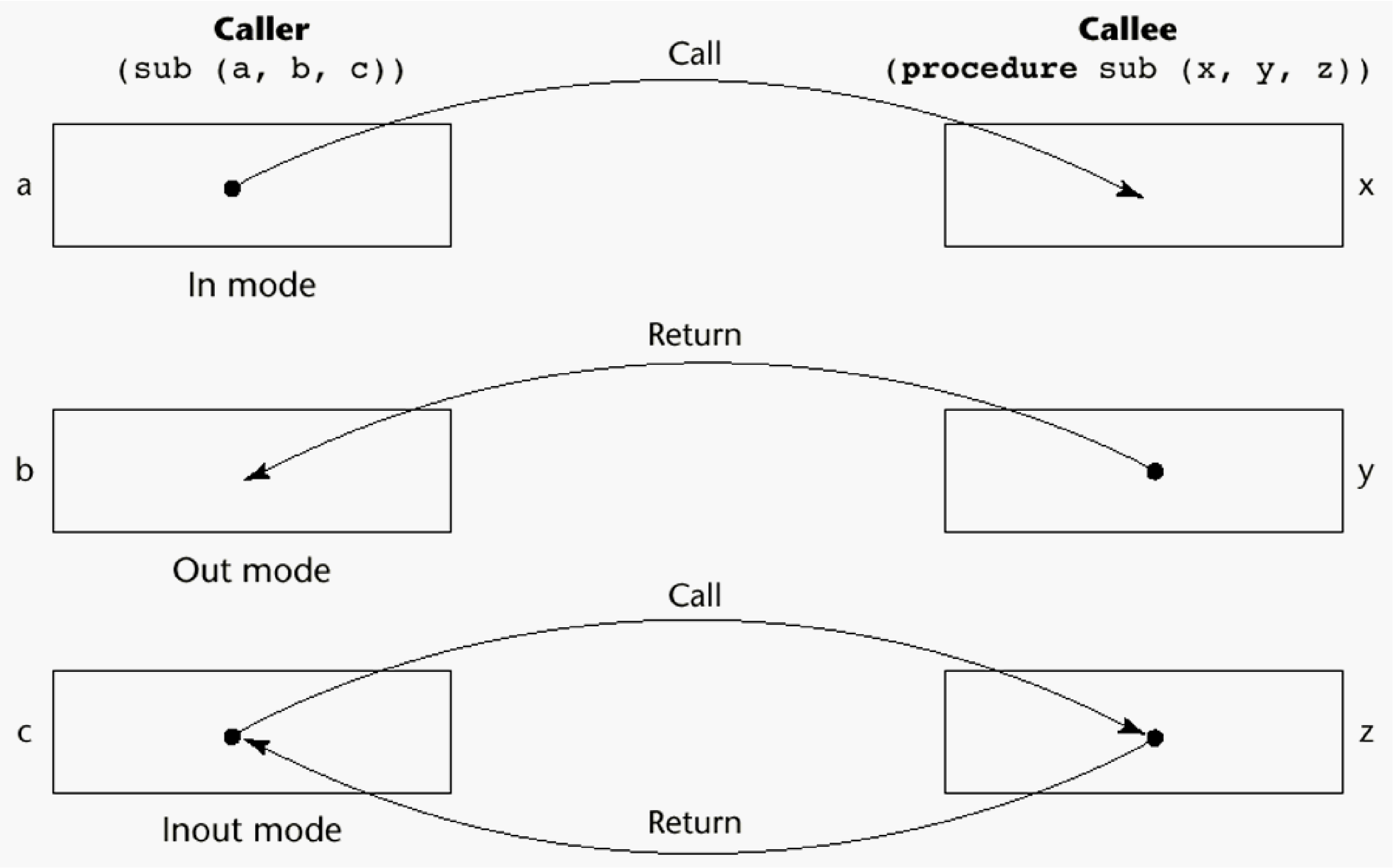
- Deliver results from the called procedure to the caller.

Input-Output Parameters

- These supply and deliver results between the caller and the called procedures.

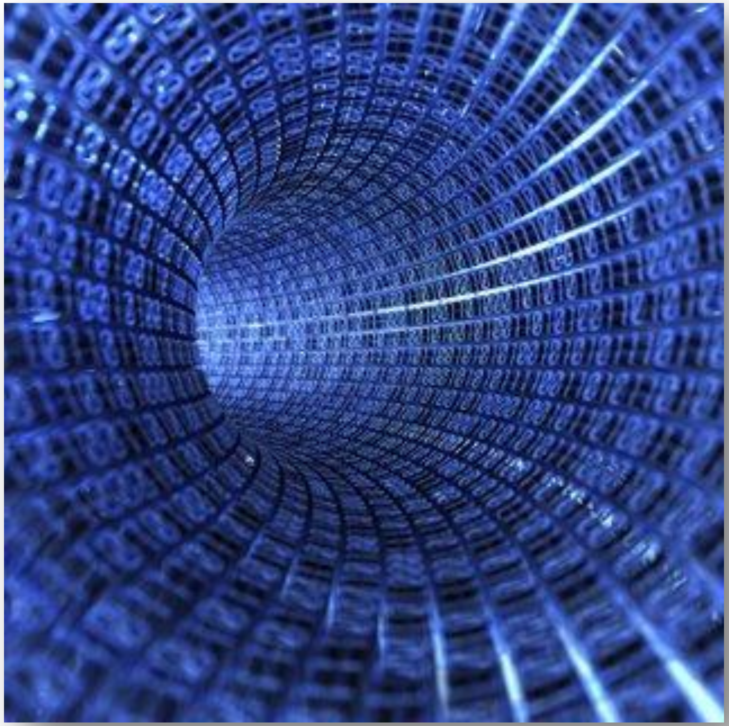


Models of Parameter Passing



Implementations of Parameter Passing

1. Call-by-Value
2. Call-by-Reference
3. Call-by-Name
4. Call-by-Return
5. Call-by-Value-Return
6. Procedures as Parameters



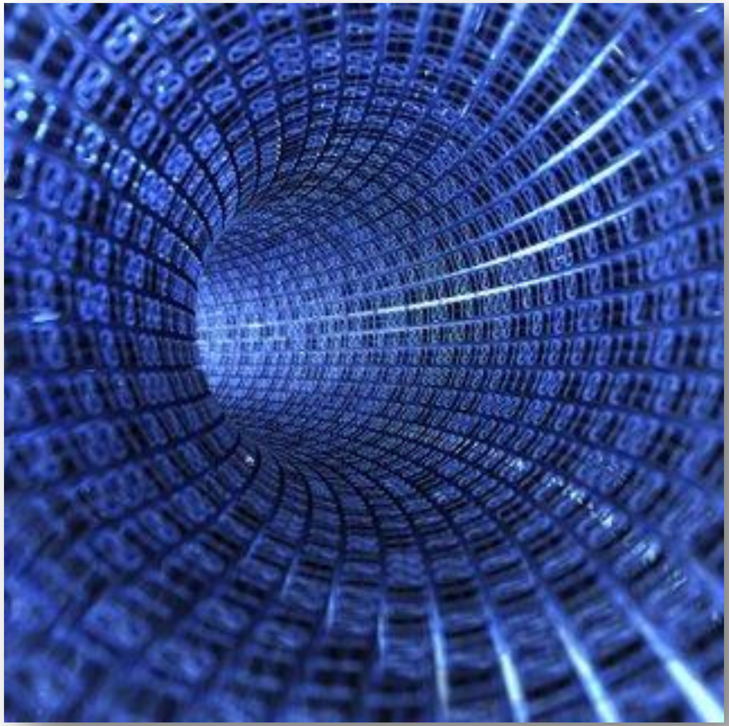
Actual Possible Implementation

Physical Move

• Copying the actual parameter from the activation record of the caller to the activation record of the callee.

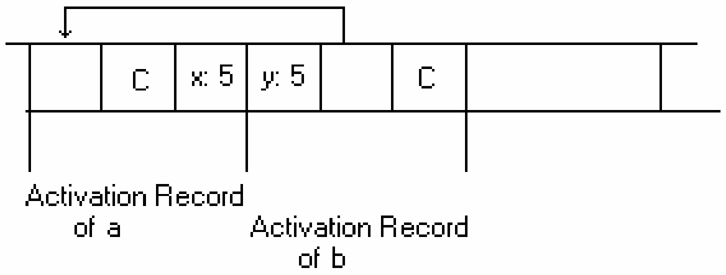
Access Path Method

• Copying the address of the formal parameter in activation record of the caller to the activation record of the callee.

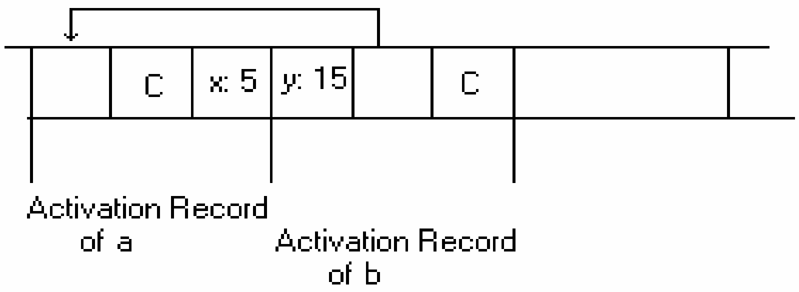


Call-by-Value

```
procedure a;
var x: integer;
  procedure b(y: integer);
  begin
      y := y + 10;
  end
begin
  x := 5;
  b(x)
  writeln(x); // Output 5
end;
```



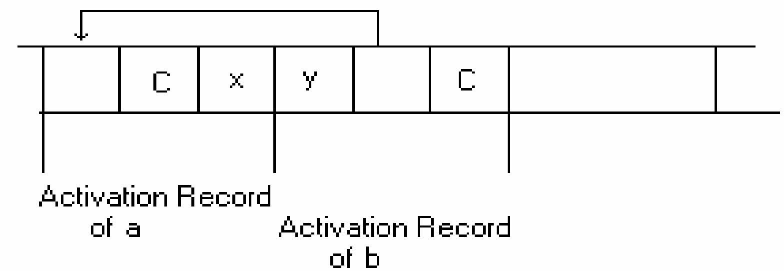
activation records before y := y + 10 is executed



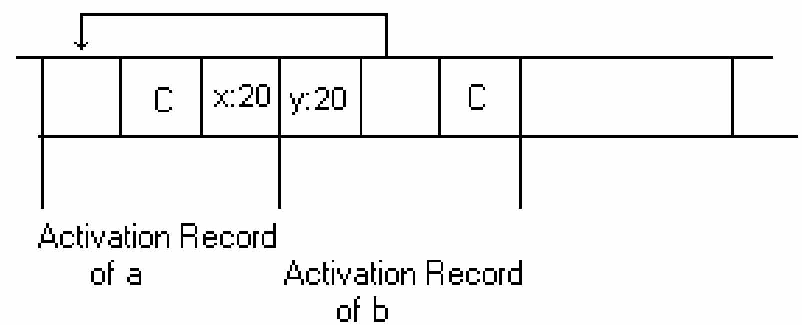
activation records after y := y + 10 is executed

Call-by-Return

```
procedure a;
var x: integer;
  procedure b(y: integer);
  begin
    y := 10;
    y := y + 10;
  end
begin
  x := 5;
  b(x); // Output 20
  writeln(x);
end;
```



stack on entry to procedure b



stack after a return from procedure b

Call-by-Value-Return

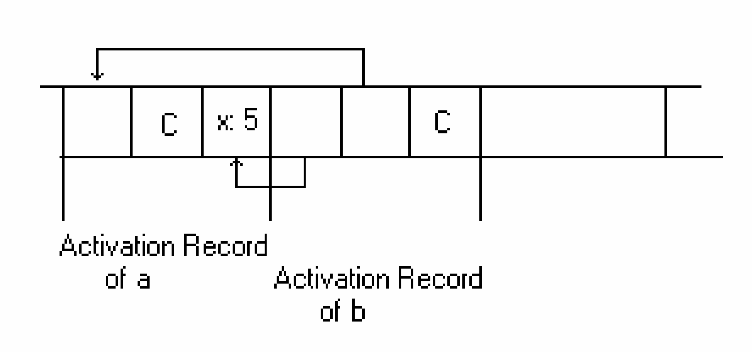
- Physical move, both ways.
- Also called **pass-by-copy**.
- A combination of call-by-value and call-by-return, hence the name.

Call-by-Reference

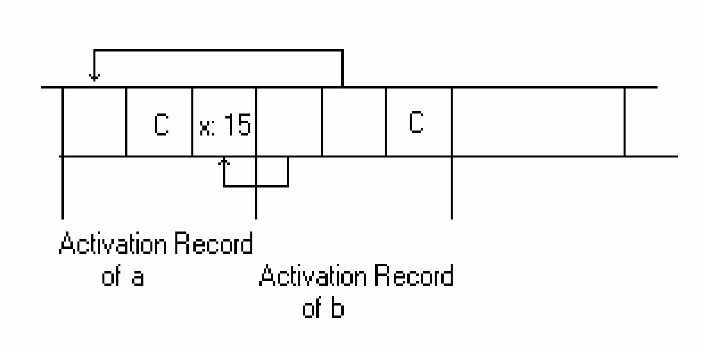
- Transmits an access path, sometimes just an address, to the called procedure.
- **Advantages**
  - ✓ Time and space efficiency of the passing.
- **Disadvantages**
  - ✓ Slower access to formal parameters compared to call-by-value.
  - ✓Collisions can occur between actual parameters.

Call-by-Reference

```
procedure a;
var x: integer;
  procedure b(var y:
   integer);
  begin
    y := y + 10;
  end
begin
  x := 5;
  b(x)
  writeln(x);
  // Output 15
end;
```



activation records before y := y + 10 is executed



activation records after y := y + 10 is executed

Chapter 8: Procedures/Subprograms

Call-by-Reference: Collision

Consider the ff C++ declaration:

```
void fun (int &first, int &second)
```

There is collision in the following instances:

- `fun(total, total)`
- `fun( list[i], list[j]) if i == j`

Call-by-Name

- By **textual substitution**.
- Formals are bound to an access method at the time of the call, but actual binding to a value or address takes place at the time of a reference or assignment.
- It was implemented for the purpose of flexibility of late binding.
- But, it is very expensive.

Call-by-Name

```
procedure a;
var x: integer;
  procedure b(var y:
   integer);
  begin
    y := y + 10;
  end
begin
  x := 5;
  b(x)
  writeln(x); // Output 15
end;
```

```
procedure a;
var x: integer;
  procedure b;
  begin
    x := x + 10;
  end
begin
  x := 5;
  b(x)
  writeln(x);
end;
```

Procedures as Parameters

- **Main implementation problem:**
  How to set its local environment.

- **Depends on the type of binding used:**
  - ✓ **Dynamic Binding**
    Environment of a procedure is not set until the procedure is called.
  - ✓ **Static Binding**
    Environment of a procedure is fixed and known at compile time; Procedure identifiers are statically bound to their names.

Procedures as Parameters

```
main(){
    int i, x, y;
    scanf("%d %d %d",&i, &x, &y);
    if (i % 2 == 0)
      printf("%d\n",winner(x,y, lower));
    else
      printf(%d\n", winner(x,y, higher));
}
int winner(a,b,decider)
//old way of declaring
int a,b;
int (*decider)();{
    return( (*decider)(a,b));
}
```

Procedures as Parameters

```
int lower(a,b)
int a, b;
{
    return( (a>b) ? b: a);
}

int higher(a,b)
int a, b;
{
    return( (a> b) ? a : b);
}
```

Possible Implementation of Procedures as Parameters

- The compiler writer must set the environment of a procedure himself.
- User is required to pass the procedure's local environment together with its entry point.
- When a procedure is called, the local environment used is the one that was passed instead of the local environment derived from the local environment of its caller.