Parallel execution of object-processes in a computer network

Oskar Świda B. Ciesielski P.Susicki

May 24, 1996

Abstract

We present a tool which permits to treat local computer network as a virtual, multiprocessor LOGLAN machine.LOGLAN is an object-oriented programming language supporting parallel execution of object-processes and equipped with an unique, truly object mechanisms for synchronization and/or communication among processes (alien call). LOGLAN programmer can explicitly allocate a object-process on any network node (there is also possibility to calculate the location). Processes at different nodes are executed parallely. This article presents the problems and solutions of object-process allocation, communication mechanisms and related questions. The presently supported platforms are a heterogenous network of Unix machines and local network of DOS machines which are communicating using TCP/IP protocol.

Contents

1 Introduction 3

2 Terminology and basic ideas 3
2.1 Terminology 3
2.2 Object-process concept 4

3 Problem specification 6
3.1 Object-process referencing 6
3.2 Initialization of LOGLAN virtual machine - assembling the machine 6
3.3 Creation and allocation of object-process on remote network node 7
3.4 Activation of a passive process (operation resume) 7
3.5 Calling method from remote object-process 7

4 Solution 9

5 How does it work ? 9

6 How to use it? 12

7 Applications and future research 13

8 Comparison to other distributed and parallel languages 14

1 Introduction

There is a few truly object-oriented programming languages which support implementation of parallel algorithms. One of them is the latest version of LOGLAN - a language implemented at Warsaw University in 1982. (The comparison to other languages we present in Chapter 8) We have obtained availability of parallel programming by parallel execution of object-processes. Processes are constructed from definition modules written by programmer (each object has its own memory and resources) [1], then they are allocated on a concrete machine and executed. Appendix A describes structure of process module and how does it work.We can treat local computer network (where the LOGLAN interpreter is working on each computer) as a virtual, multiprocessor LOGLAN's machine. Network node becomes then a virtual LOGLAN processor. Additionally such processor is able to execute several processes concurrently, so we have parallel and concurrent programming together!
This article presents ideas and implementation of allocation and communication mechanisms for object-processes.

As a conclusion we can say that there is a posibility to construct object-oriented programming language, where object-processes are executed in parallel.As a result we obtained a very cheap multiprocessor system which cost is equal to several computers connected by local network (such network usually exist, and there is only need to install LOGLAN interpreter on each computer).

2 Terminology and basic ideas

For better understanding this paper we present used terminology and idea of object-process.

2.1 Terminology

LOGLAN object-oriented programming language implemented at Warsaw University in 1982 and then developed LITA, Université de Pau and at Bialystok University of Technology in 1995. For more information about LOGLAN language see at LOGLAN repository: *http://aragorn.pb.bialystok.pl*

TCP/IP communication protocols from DoD family (Transmission Control Protocol and Internet Protocol), used to provide communication mechanisms in local computer network.

Interpreter program which interprets LOGLAN code

Network node workstation in a local computer network.

[1] shared memory problem will be researched later

Console node where the main program block is executed.

Object-process process defined in LOGLAN language

Alien call communication mechanism developed by B.Ciesielski

Virtual LOGLAN machine set of interpreters working at nodes in local computer network

Virtual LOGLAN processor network node with LOGLAN interpreter working on it.

2.2 Object-process concept

LOGLAN language offers a special objects for use in concurrent and parallel programming. These are object-processes. We use that name to point out differences between UNIX processes and LOGLAN processes. Object-process (called later O-process) is an object and a process simultaneously. In "process aspect" it is simply a thread, "object aspect" however gives him all object properties like:

- inheritance
- methods and data
- encapsulation

Moreover LOGLAN processes can dynamically decide which of their methods are available for another objects. Privacy and publicity of given method can change in time. Object-process can mark that method is private or public during process execution. Each O-process has an "access mask" - the set of procedure names which are available for external call. O-process can add or remove name to/from the set causing procedure to be enabled (public) or disabled (private) respectively.

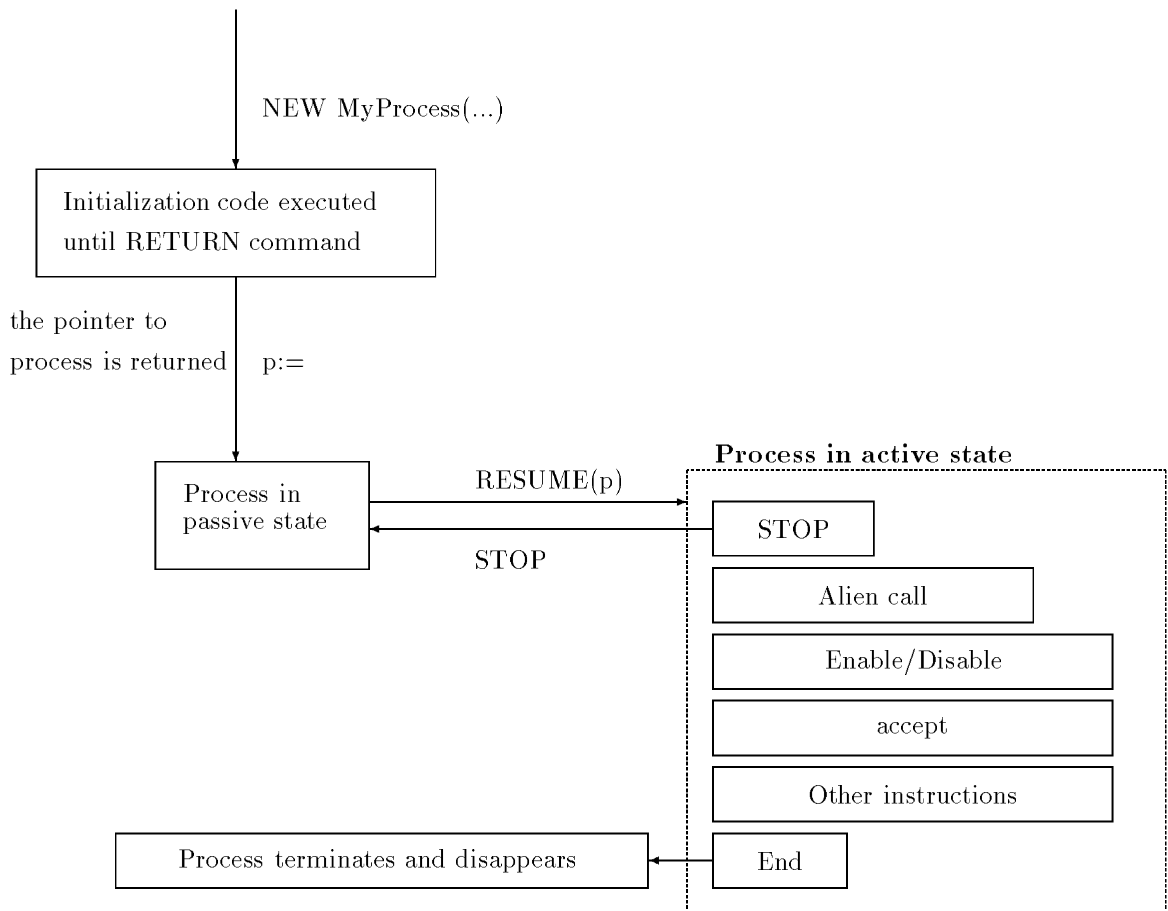
Below we present definition module of O-process and its working scenario:
Definition module syntax:

unit *module name* **: process(***parameters***)**
local declarations
begin
initialization code;
return;
process code
Here we can put: classical instructions, **stop**, **resume(...) call**, **enable/disable**, **end**, **accept**
end*module name*

Working scenario:
We assume that process is created by instruction:

p := new MyProcess(...)

Picture below presents the scenario:



3 Problem specification

During the research we have tried to answer a question:

Are we able to implement a system for parallel execution of LOGLAN object-processes?

The main idea for resolving specified problem lies in constructing (and implementing) LOGLAN Virtual Machine. Each network node participating in program execution is treated as a virtual processor. Each virtual processor works independently but is able to communicate with another processors. Programmer can allocate O-processes on any virtual processor. Such machine enables parallel execution.

We assume some facts:

- All computers are DOS or UNIX machines able to communicate using the TCP/IP protocol.
- Network is reliable, that means there is no danger that it will stop transmiting messages during program execution
- There are LOGLAN interpreters installed on each network node we want to use.

The problem was divided into five parts:

3.1 Object-process referencing

As we mentioned above, programmer can allocate O-process on any network node. Of course each allocated O-process must be available by reference for future use in program. The question is how should we implement O-process reference to be able to referencing processes on remote network nodes. We implement reference as a union of two informations: number of the virtual processor on which O-process is allocated and memory reference (on that processor) where the process code exists.

3.2 Initialization of LOGLAN virtual machine - assembling the machine

LOGLAN virtual machine consists of computers connected to local network with interpreters working on them. Single computer is called *virtual LOGLAN processor*. We distinguish one of these processors as a *console*, it will be executing

the main program block. At the initial state the console will construct connection to each node we want to use during program execution and check if such a connection works properly. This assures that the interpreters can communicate to each other.

3.3 Creation and allocation of object-process on remote network node

Allocation of process P2 on node Y made as a result of request sent from node X by process P1 one can present following way:

- process P1 sends request to allocate process P2 on node Y
- interpreter on node X sends this request to interpreter on node Y
- interpreter on node Y builds local instation of P2 from definition module, executes the initialization code and returns P2 pointer to the interpreter X
- the interpreter X sends P2 pointer to P1.

The main goals of implementation are:

1. sending allocation request to proper node
2. returning (and receiving) acknowledge of allocation

3.4 Activation of a passive process (operation resume)

Activation of process P2 is made on explicit request from any other process (for example P1) by executing command:

resume (P2)

in the code of the process P1. Such an operation needs neither acknowledge nor process synchronization because refers only to activated process. Implementation task here is to send resume request to proper node.

3.5 Calling method from remote object-process

Majority of languages for parallel (distributed) programming use the Remote Procedure Call mechanism for communication. We could use this , but B.Ciesielski developed unique mechanism called "alien call".
Figure 1 presents the main idea of "alien call". In order to call procedure from remote O-process this process must be in active state and called procedure

must belong to access mask. General idea is similar to RPC but B.Ciesielski extended this by constructing two forms of the call which we named "asynchronous" (Figure 2) and "synchronous" (Figure 3) calls. "Asynchronous" call simply interrupts P2 and forces execution of proc1. "Synchronous" is similar to rendez-vous mechanism. Which kind of call occurs depends on state of P2. If P2 executes **accept** command before P1 calls proc1 it's "synchronous" call, if P1 calls proc1 and P2 is not waiting for that procedure (P2 is simply running his own code) then the "asynchronous" version of the call takes place. In both cases it is required that proc1 is enabled.

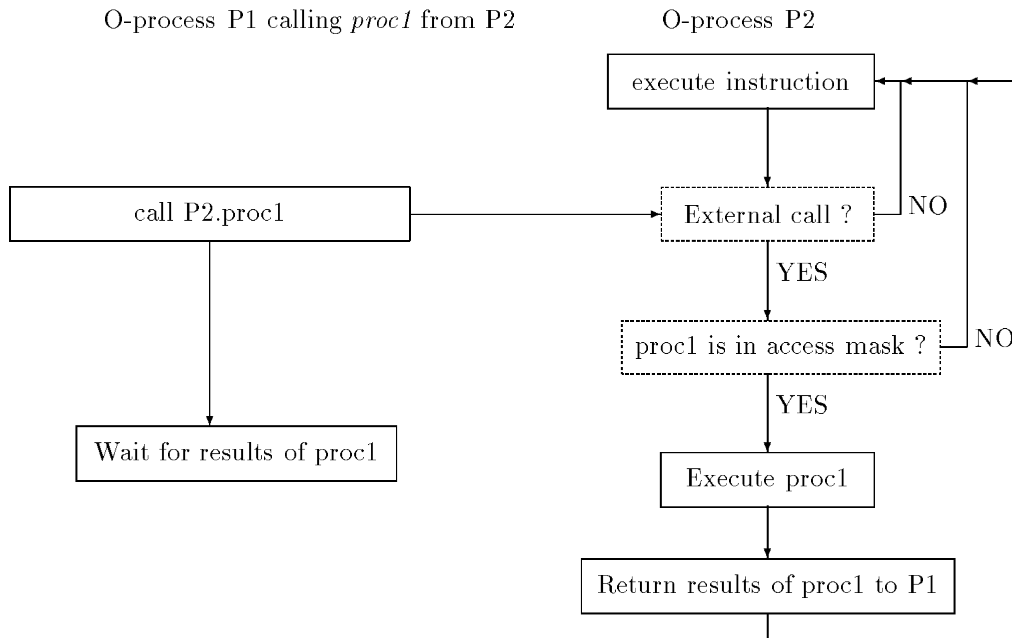


Figure 1: The main idea of "alien call"

4 Solution

Implementation and research have proved that the answer to the question we asked in the chapter 3 is YES.

MAIN RESULT: **We constructed the LOGLAN Virtual Machine which is able to execute O-processes parallely and realizes all five requirements mentioned in Chapter 3**

Our solution realizes multiprocessor paralellism through special protocols that enable allocation and communication of processes.

Allocation The programmer may decide where the code of the specific process-object should be executed. There is also posibility to compute, which node should be used so one can implement any algorithm of automatic allocation.

Communication Allocation of processes and communication between them is based on message exchanges.

Parallelity Each interpreter has own process tables and execute them independently to other nodes what gives us required parallelity. The main program block is proceeded on console.

Multiprocessor We treat our solution as a multiprocessor, virtual machine.

Protocols All of communication is performed using only four types of protocols:

1. initial connection - assembly of a virtual LOGLAN machine
2. process allocation
3. resume request
4. alien call

The communication scenarios for these messages are described in next chapter.

5 How does it work ?

As we noticed before there is four scenarios of communication. Let's present them:

Console interpeter	*Node #1 interpreter*	...	*Node #n interpreter*
Wait for n calls from clients	Connect to console and wait for acknowledge	...	Connect to console and wait for acknowledge
If you have received all calls then register them and send acknowledges to all clients	If you have received acknowledge then start wor- king - connection established	...	If you have received acknowledge then start wor king - connection established

1. **Initial phase** - we cosider making connection between console and n network nodes.

2. **Process-object allocation** - process Z working on network node #W request allocation of process Y on network node #X.

Interpreter on node #W	*Interpreter on node #X*
If you have received allocation request send this request to node #X and let the process Z wait for CREACK	
	If you have received message CREATE(Y) then construct an Y instance from definition module, execute initialization code and send pointer and CREACK to node #W
If you have received CREACK message (and pointer with it), activate process Z and return him pointer.	

It's easy to notice that if CREACK message won't reach network node #W (due to transmission error) process Z will never be activated! Thus the assumption of network availability is very important.

3. **Resume operation** - process Z on node #W activates process Y on node #X

Interpreter on node #W	*Interpreter on node #X*
If you have received resume request then send it to node #X	
	If you have received resume message then activate process Y

As we mentioned, such operation doesn't need an acknowledge.

4. **Alien call**
   Here we present two versions of communication scenario because alien call may be considered as a asynchronous or synchronous dialog. We consider that process Z working on node #W is calling procedure *proc1* from process Y working on node #X.
   - asynchronous version (means Y is active and there is *proc1* in his access mask)

Interpeter on node #W	*Interpeter on node #X*
If you have received call request then send RPCCALL(Y,proc1) message to node X and stop process Z	
	If you have received RPCCALL(Y, proc1) then: - check access mask - stop process Y - save access mask and clear it - execute proc1
	Here the proc1 code is executed
	If proc1 execution is complete then send results and RPCACK message to node #W
If you have received RPCACK (and results of proc1) then activate process X and return him results.	Restore access mask and activate process Y

   - synchronous version (Y is active, there is proc1 in access mask,and Y executes *accept* instruction)

Interpeter on node #W	*Interpeter on node #X*
If you have received call request then send RPCCALL(Y,proc1) message to node #X and stop process Z	Execute *accept* instruction and wait for external call
	If you have received RPCCALL(Y, proc1) then: - check access mask - execute proc1
	Here the Y code is executed
	If proc1 execution is complete then send results and RPCACK message to node #W
If you have received RPCACK (and results of proc1) then activate process X and return him results	Restore access mask and continue process Y after *accept*

6 How to use it?

Now we describe the actions which should be made when you want to use parallel programming. In order to assemble and initialize a LOGLAN virtual machine one should execute the **int** command on every node which is concerned (this may be done by a remote script also). There are two assumptions:

- Each network node used by programmer is given a unique number from 0 to 255 (during process-object allocation programmer use that number as an virtual processor id)
- Network node number 0 is a special node (console) where the main program block is executed.

If one wants to run a parallel program (let it be program written in file named *prog*) you must run interpreters on each node you want to use. Starting interpreter on specific node you explicitly choose the number for this node. After checking the connections the interpreter on console starts to execute the main block. Let's take an example:
Assume that programmer allocates two processes P1 and P2 on nodes #3 and #5. So he use three computers in that case : #0 (console) , #3 and #5. The main block will be executed on console and processes P1, P2 on proper nodes. To run execution you should execute:
a) on the node which you want to be a console: **int -r 0** *number_of_clients filename* (int -r 0 2 prog)
A -r option says you want to distribute your processes, 0 describe this node as a console and 2 means that interpreter should wait for two clients
b) on the other nodes execute: **int -r** *client_number console_node_address:port filename* (int -r 3 aragorn:3600)

client_number means number which you give for node (in that case it will be 3 and 5 respectively)
console_node_address is an internet address of computer which you have decided to be a console,
port is a server port for the all interpreters (default is 3600)

7 Applications and future research

We can point on three main applications of that version of LOGLAN programming language:

1. A cheap multiprocessor machine
   Multiprocessor machines are quite expensive, but now you can built your machine using your local computer network.
2. Truly object-oriented distributed and parallel programming
3. Didactical purposes

LOGLAN language may be an excellent for studying of parallel and distributed algorithms.
Implemented version is not a final product. We want to implement parallel execution of process-objects on DOS machines using again a TCP/IP protocol and maybe IPX protcol. New process conception is also a starting point to further researches, perhaps some modifications to LOGLAN language should be made. We could also consider many network problems related to network relaibility (for example reactions to network errors, real time systems etc.)

8 Comparison to other distributed and parallel languages

There are many languages which support distributed and parallel programming. Few of them are truly object-oriented (PVM, PCN, ConcertC are not at all). We can compare only two of them: Synchronizing Resources and Emerald. The comparison is shown below:

Property	*Synch. Resources*	*Emerald*	*LOGLAN*
Classes	Yes	No	Yes
Inheritance	Yes	No	Yes
Private/Public methods	No	Yes	Yes
Dynamic changes of method privacy/publicity	No	No	Yes
Persistent objects	No	Yes	No
Communication mechanism	RPC and rendez-vous	?	alien call
Object location independence	No	Yes	Yes
Object mobility	No	Yes	No
Works on UNIX machines	Yes	Yes	Yes
Works on DOS machines	No	No	Yes

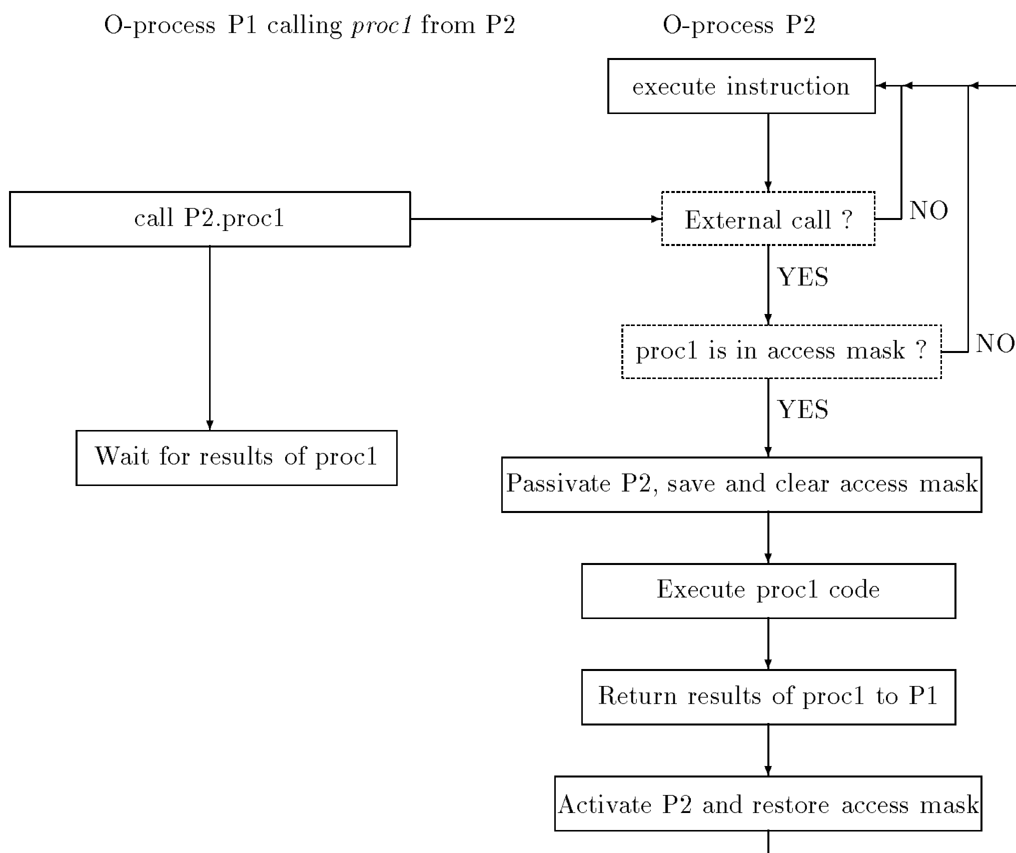


Figure 2: "Asynchronous" version of "alien call"

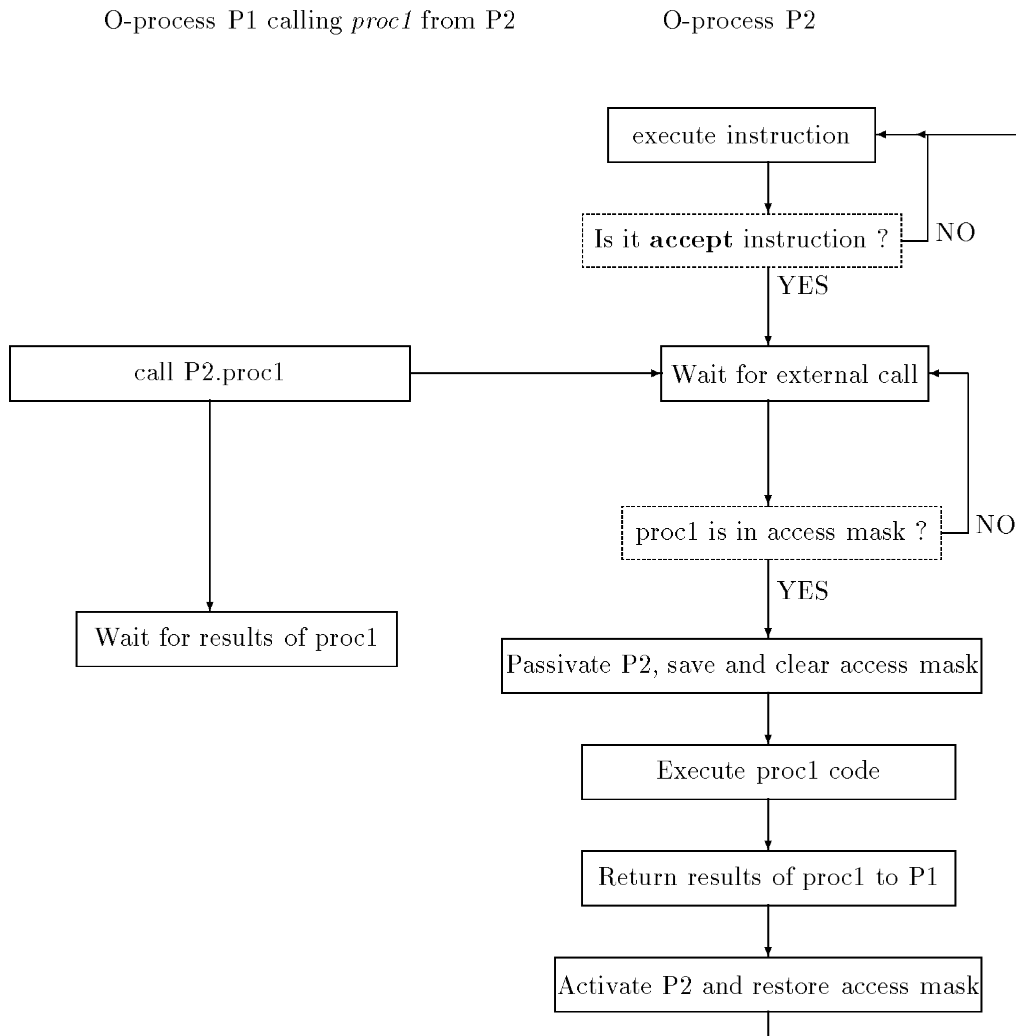


Figure 3: "Synchronous" version of "alien call"