TR-0764

Resource Management Machanism of PIMOS

by
H. Yashiro, T. Fujise, T. Chikayama, M. Matsuo,
A. Hori (Mitsubishi) & K. Wada

April, 1992

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome

(03)3456-3191 ~ 5
Telex ICOT J32964
Minato-ku Tokyo 108 Japan

Institute for New Generation Computer Technology

Resource Management Mechanism of PIMOS

Hiroshi YASHIRO*†, Tetsuro FUJISE‡, Takashi CHIKAYAMA†,
Masahiro MATSUO‡, Atsushi HORI‡ and Kumiko WADA†

† Institute for New Generation Computer Technology
1-4-28 Mita, Minato-ku, Tokyo 108, Japan
‡ Mitsubishi Research Institute, Inc.
1-8-1 Shimomeguro, Meguro ku, Tokyo 153, Japan

Abstract

The parallel inference machine operating system (PIMOS) is an operating system for the parallel inference systems developed in the Japanese Fifth Generation Computer Systems project. PIMOS is written in a concurrent logic language KL1, which adds numerous extensions to its base language, Guarded Horn Clauses, for efficient meta-level execution control of programs. Using such features, PIMOS is designed to be an efficient, robust and flexible operating system. This paper describes the resource management mechanism of PIMOS, which is characterized by its unique communication mechanism and hierarchical management policy.

Hierarchical management of user tasks in a distributed fashion is mandatory in highly parallel systems so that the management overhead of the operating system can also be distributed to the processors running in parallel. The meta-level execution control structure, called shoen, is provided by the KL1 language and is used for providing such hierarchical management in a natural fashion.

In concurrent logic languages, message streams implemented by shared logical variables are frequently utilized the media of interprocess communication. PIMOS, based on this programming style, provides multiplexed streams with flexible control for communication between user programs and the operating system.

1 Introduction

In the Fifth Generation Computer Systems project of Japan, the parallel inference machines, PIMs, have been developed to provide the computational power required for high performance knowledge information systems [Goto *et al.* 1988, Taki 1992].

The parallel inference machine operating system, PIMOS [Chikayama *et al.* 1988], was designed to control highly parallel programs efficiently on PIMs and provide a comfortable software development environment for concurrent logic language KL1.

PIMOS was first developed on an experimental model of parallel inference machine, called Multi-PSI [Nakajima *et al.* 1989], consisting of up to 64 processing elements connected via a two-dimensional mesh network. The system was first developed in 1988 and has been used since then to research and develop various experimental parallel application software. Later, the system was ported to several models of parallel inference machines with considerable improvements in various aspects.

*EMAIL : yashiro@icot.or.jp

1.1 Shoen Mechanism

The language in which PIMOS and all the application programs are written is called KL1. KL1 is a concurrent logic language based on Guarded Horn Clauses [Ueda 1986] with subsetting for efficient execution and extensions for making it possible to describe the full operating system in it.

The greatest benefit of using a concurrent logic language in writing parallel systems is the implicit concurrency and data-flow synchronization features. With these features, one of the most difficult parts of parallel programming, synchronization, becomes automatic, making software development much easier than in conventional programming languages with explicit synchronization.

An important addition by the KL1 language to regular concurrent logic languages is its meta-level execution control construct named shoen. Shoen enables the encapsulation of exceptional events and the description of explicit execution control over a group of parallel computational activities. The execution unit of KL1 programs is a preposition called a goal, which will eventually be proven by the axiom set given as the program. This proof process is the execution process of the programs, as it is with any other logic programming languages. As the proof process can proceed concurrently for each goal, the goals are fine-grained parallel processes.

As no backtracking feature is provided in concurrent logic languages, all the goals in the system form one logical conjunction. Thus, if no structuring mechanism is available, failure in a user's goal means failure of the whole system. The shoen mechanism provides a way of grouping goals, isolating such failure to a particular group of goals. Such a group is called a shoen.[1]

A shoen can be initiated by invoking the following primitive.

```
execute(Code, Argv, MinPrio, MaxPrio, ExcepMask, Control, ^Report)
```

The arguments `Code` and `Argv` represent the code and arguments of the initial goal of the shoen. This goal is reduced to simpler goals during the execution (or proof) process, and all such descendant goals will belong to this shoen.

A shoen has a pair of streams named the *control stream* and the *report stream*, which are represented here by the two arguments `Control` and `Report` respectively. The control stream is used to send commands to control the gross execution of the goals belonging to the shoen, such as starting, stopping, resuming or aborting them as a group. Exceptional events internal to the shoen, such as failure, deadlock, exception such as arithmetical overflow, or termination of computation are reported by the messages received from the report stream (Figure 1).

The two arguments `MinPrio` and `MaxPrio` specify the priority range of the goals belonging to the shoen. PIMOS does not try to control scheduling of each fine-grained parallel process, but controls them as a group using the control stream and this priority mechanism.

Shoen can be nested by arbitrary levels. Stopping a shoen, for example, will make all the children or grandchildren shoen inside it. The argument `ExcepMask` is used to determine which kinds of exceptional events should be reported to this particular level of the hierarchical structure of the shoen.

PIMOS supervises user programs using this shoen mechanism. The exception reporting

[1] The shoen mechanism is an extension of the meta-call construct of Parlog [Foster 1988] and can be considered to be a language-embedded version of the meta-interpreters seen in systems based on Concurrent Prolog [Shapiro 1984]

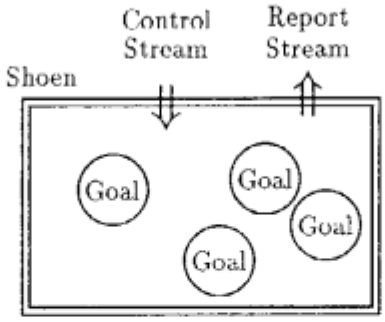


Figure 1: Shoen

mechanism is used to first establish the communication path from the user programs to PIMOS. An exceptional event to be reported can be intentionally generated using the following primitive.

```
raise(Tag, Data, Info)
```

The argument `Tag` specifies the kind of event generated by this primitive. This, along with the mask specified when the shoen is created, determines at which level in the shoen hierarchy this event should be processed.

The two arguments `Data` and `Info` are passed as detailed information of the event. The `Data` argument can be any data, instantiated, uninstantiated or partly instantiated, while the `Info` argument has to be instantiated before the event is generated. The above primitive will be suspended until this argument is completely instantiated to be a ground term without any logical variables.

By monitoring the report stream, PIMOS can receive the requests from the user as messages coming from the stream in the following format.

```
exception(Kind, EventInfo, ^NewCode, ^NewArgV)
```

The `Kind` argument indicates the kind of exceptional event. In this case, the fact that the event was intentionally generated can be recognized.

The `EventInfo` argument is more detailed information of the event. In the above case, the `Data` and `Info` arguments of the `raise` primitive will be combined together through this argument.

The `NewCode` and `NewArgV` arguments specify an alternative goal to be executed in the object level in place of the goal that generated the event. PIMOS utilizes such a goal for inserting a protection filter, which will be described later.

1.2 Resources

In conventional systems, memory management and process management are two of the most important tasks of the operating system. In the case of PIMOS, as the underlying language implementation of KL1 provides primitives for those fundamental resources, PIMOS do not have to be concerned with such low-level management.

KL1 provides automatic memory management feature including garbage collection, as is the case with Lisp or Prolog. Thus, basic memory management is automatic in the language implementation. KL1 provides implicit concurrency and data-flow synchronization, context switching or scheduling is already supported by the language. Thus, PIMOS

does not deal with low-level fine-grained process management, but controls larger-grained groups of processes using the priority system provided by the language.

As memory and process are managed in the KL1 language implementation level, we call them *language-defined resources*. On the other hand, other higher-level resources, such as virtual I/O devices, are more directly controlled by PIMOS. We call them *OS-defined resources*. In what follows, we will concentrate on the management of such OS-defined resources.

2 Communication Mechanism

The basic principles of the communication mechanism are described in this section. This lays the basis for the foundation of the PIMOS resource management mechanism.

2.1 Stream Communication

In a parallel environment, efficient management of various resources becomes much more difficult than in a sequential environment. When data in a particular memory area should not be overwritten while being processed by the operating system, the operating system can simply suspend the execution of user programs in a sequential system. In a highly parallel environment, this will seriously spoil the merit of fine-grained parallelism, as all the user processes sharing the memory space must be stopped irrespective of whether they actually have any possibility of changing the data.

A frequently used programming technique in concurrent logic languages is the object-oriented programming style [Shapiro and Takeuchi 1983]. In this style, a process (actually a goal which becomes perpetual by recursively calling itself) can have internal data which cannot be accessed from outside and shared data containing variables which can be used for interprocess communication. Interprocess communication is effected by gradually instantiating the data shared between processes. Instantiation corresponds to sending data and observing it corresponds to receiving the data. When the shared data is instantiated gradually to a list structure of messages, the structure can be considered to be a communication stream. PIMOS also utilizes this technique for communication between the user programs and the operating system.

For example, reading a character string from the keyboard can be effected by a program shown in Figure 2 (after establishing a communication path by generating an exceptional event as explained in a previous section). The user sends a message `getb/2`, that requests the reading of `N` characters. When PIMOS receives the message, it reads `N` characters from the keyboard to the variable `KBDString` (`readFromKBD/2`). Then, the user receives the `String` instantiated to `KBDString`. As the cdr of the list, `ReqT`, will be a new shared variable after this operation, it can be used for successive such communication.

2.2 Protection Mechanism

In a system based on a concurrent logic language, many of the problems that might arise in a conventional operating system will never be a problem. As the communication path between the user programs and the system programs can be restricted to shared logical variables, there is no way for user programs to overwrite the memory area used by the system programs.

```
?- pimos(Req), user(Req).

user(Req) :-
    true |
        Req = [getb(N,String)|ReqT],
              ......      .

pimos([getb(N,String)|ReqT]):-
    true |
        readFromKbd(N,KBDString),
        KBDString=String,
        pimos(ReqT).
```

Figure 2: An example of interprocess communication between user and PIMOS

With the simple mechanism described above, however, intentional or accidental error in user programs may cause system failure in the following ways.

Multiple Writer Problem When both the system and user programs write different values to the same variable, a unification failure may occur. In a concurrent language like KL1, unifications by PIMOS and the user may be executed concurrently. Thus, this contradiction may cause PIMOS to fail if it tries to instantiate the variable later.

Forsaken Reader Problem The user program may fail to instantiate the arguments of the message sent to PIMOS, in which case PIMOS may wait forever for it to be instantiated.

To solve problems, a filtering process called the protection filter is inserted in the stream between PIMOS and the user program. This filter is inserted in the object-level (within the user's shoen) using the above described `NewCode` and `NewArgV` arguments of the exception reporting message. To solve the forsaken reader problem, the filter will not send a message to PIMOS until its arguments are properly instantiated. To solve the multiple writer problem, the filter will not unify the result from the operating system with the variable supplied by the user until it is properly instantiated by the operating system (Figure 3).

```
filter([get(C)|User],OS):-
   true |
      OS = [get(C)|OS1],
      wait_and_unify(C1,C),
      filter(User,OS1).
wait_and_unify(OSV,UserV) :-
   wait(OSV) |
      UserV = OSV.
```

Figure 3: An example of the protection filter

In the actual implementation, such filtering programs are automatically generated from the message protocol definitions.

2.3 Asynchronous Communication

Stream communication is simple, yet powerful enough for simple applications, but it does not provide sufficient flexibility and efficiency at the same time when controlling various I/O devices.

As communication delay is a crucial factor in distributed processing, it is desirable to send messages in a pipelined manner for better throughput. To allow this, it is desirable to allow messages to be sent before being sure that they are really needed and to allow them to be canceled if they are found to be unnecessary afterwards. If only one communication stream is available between the operating system and the user, this cancellation is not possible (Figure 4).

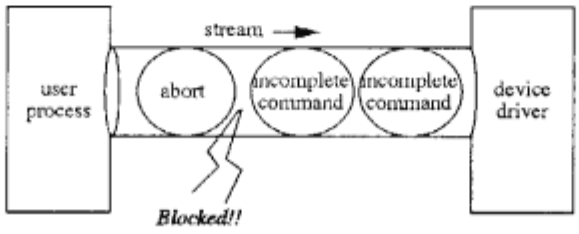


Figure 4: Blocked stream

To solve the problem, PIMOS provides another communication path for emergencies. We call the path *abort line*. This communication path is implemented as a simple shared variable. Instantiation of this variable notifies cancellation of commands already sent to the stream.

Another problem is that, with only one communication stream from the user to the operating system, there is no way for the devices to send asynchronous information to the users. To solve this, besides the above-mentioned two communication paths, a communication path in the reverse direction called the *attention line* is provided (see Figure 5).

These two "lines" are one-time communication paths in their nature. After they are used, new paths can be established by sending the reset message described below through the main communication stream.

2.4 Multiplexing Communication Paths

It is sometimes mandatory to share some (virtual) resources among several processes. A typical example is with the terminal device shared among processes running under a shell. In such cases, only one process should be able to use the device at a time, but quick switching among processes (when a process is suspended by a terminal interrupt, for example) is essential for comfortable operation. On the other hand, the pipelining of I/O request messages is mandatory for better throughput. With only the mechanism of the "abort" and "attention" lines mentioned above, the aborted requests will merely

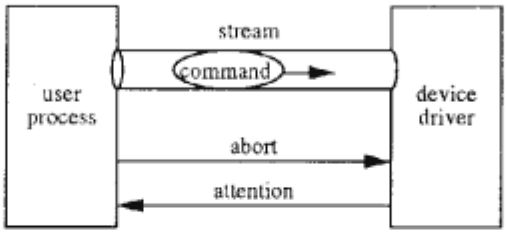


Figure 5: Asynchronous communication with a device

disappear. This does not provide more flexible control, such as suspending a process and resuming it afterwards.

PIMOS provides the following I/O messages to solve the problem.

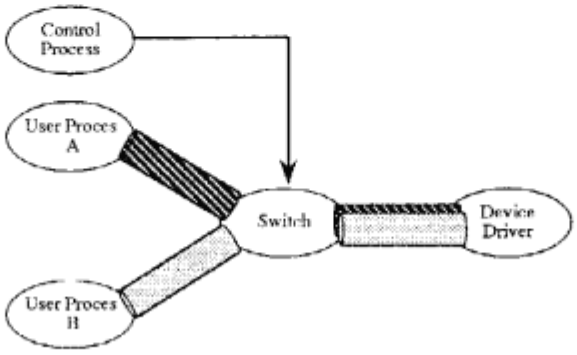
reset(^Result): The variable `Result` is instantiated to a term `normal( ^Abort, Attention, ID)`. The arguments `Abort` and `Attention` correspond to new abort and attention lines. An identifier for a sequence of commands subsequently sent on this stream is returned in the argument `ID`.

resend(ID, ^Status): When I/O request messages are aborted using the abort line, the device drivers remember the aborted messages associated with the identifier. The `resend` command tells the device driver to retry the aborted messages associated with `ID`.

cancel(ID, ^Status): This `cancel` message tells the device driver to forget about the aborted messages associated with `ID`.

Suppose that a certain device, such as a window device, is shared by two user processes, A and B. Each user process has one communication path to the device. The communication paths connected from the user processes are merged to a "switch" process, which has another communication path connected to a "control" process (Figure 6(a)).

The control process is usually a part of a program such as a command interpreter shell that lets two or more programs share one display window. When a program running under the shell is suspended by an interruption, there may remain I/O messages that have been already sent from the interrupted program to the device driver but have not been processed yet. In such a case, the control process suspends the processing through the abortion line and sends a reset message to the device through the switch process (Figure 6(b)). The suspended messages are kept in the device driver with `ID`. If the program resumes communication with the device, the control process commands the switch process to send a resend message with `ID` as its argument to make it resume the suspended I/O requests.



(a) switch for multiplexing streams

```
              abort                reset,resend
Process A -------.....................----------
            ID = 1                      ID = 1
               reset, resend    abort
Process B .......------------------.........
                      ID = 2

Example: --- : connected communication path
         ... : disconnected communication path
```

(b) commands between the switch and the device driver

Figure 6: Multiplexing streams

3 Resource Management Mechanism

All the devices provided by PIMOS have the stream interface described above, with attention and abort lines when required. Thus, management of resources in PIMOS is management of these communication paths. This section describes the mechanism of the management by PIMOS.

The following are the keywords to understand the mechanism.

Task: Tasks are the units of management of user programs. A task consists of an arbitrary number of goals (fine-grained processes) corresponding to a shoen in the language level, and forming a hierarchical structure.

General Request Device: The general request device is the top level service agent. This is the stream user programs can obtain directly from PIMOS. Request streams to all other devices are obtained by sending messages to this device.

Standard I/O Device: A task is associated with its standard I/O devices. Standard I/O devices are aliases of some devices they are associated with. The correspondence is specified when the task is generated. The resource sharing mechanism described above is attached to these tasks.

Server: I/O subsystems of PIMOS are actually provided by corresponding tasks called *servers*. They are made relatively independent of the kernel of PIMOS, making the modularity of the system better. The file subsystem is typical of such servers.

3.1 Resource Management Hierarchy

As mentioned above, *tasks* are the unit of management of user programs. All communication paths from user program to PIMOS are associated with certain tasks. Resources obtained by requests through such paths are also associated with the tasks.

Tasks are implemented using the shoen mechanism of KL1. A task is a shoen with its supervisor process inside the PIMOS kernel. The kernel controls the utilization of resources within the task.

Tasks are handled just like ordinary I/O devices. A task handler is a device handler whose corresponding device happens to be a shoen. Tasks are unique in that they may have children resources. As its consequence, a task can have tasks as its children resources forming a nested structure. Corresponding to this, task handlers and other resource controlling processes inside PIMOS also form a hierarchical structure, called the *resource tree*. This resource tree is the kernel of resource management by PIMOS.

One layer of the resource tree is represented by the task handler and *device monitors* corresponding to its children resources connected by streams in a loop structure (Figure 7). Device monitor processes are common with all kinds of devices. Associated with each device monitor is a *device handler*, which depends on the category of the device. Device monitors and device handlers are dynamically created when a new virtual device is created and inserted in the loop structure.

The device handlers can be classified as follows.

Task Handler: A task handler corresponds to a shoen. As described above, usual shoens whose control and report streams are directly connected to their creator. Those streams of shoens corresponding to a task are connected to the task handler. The

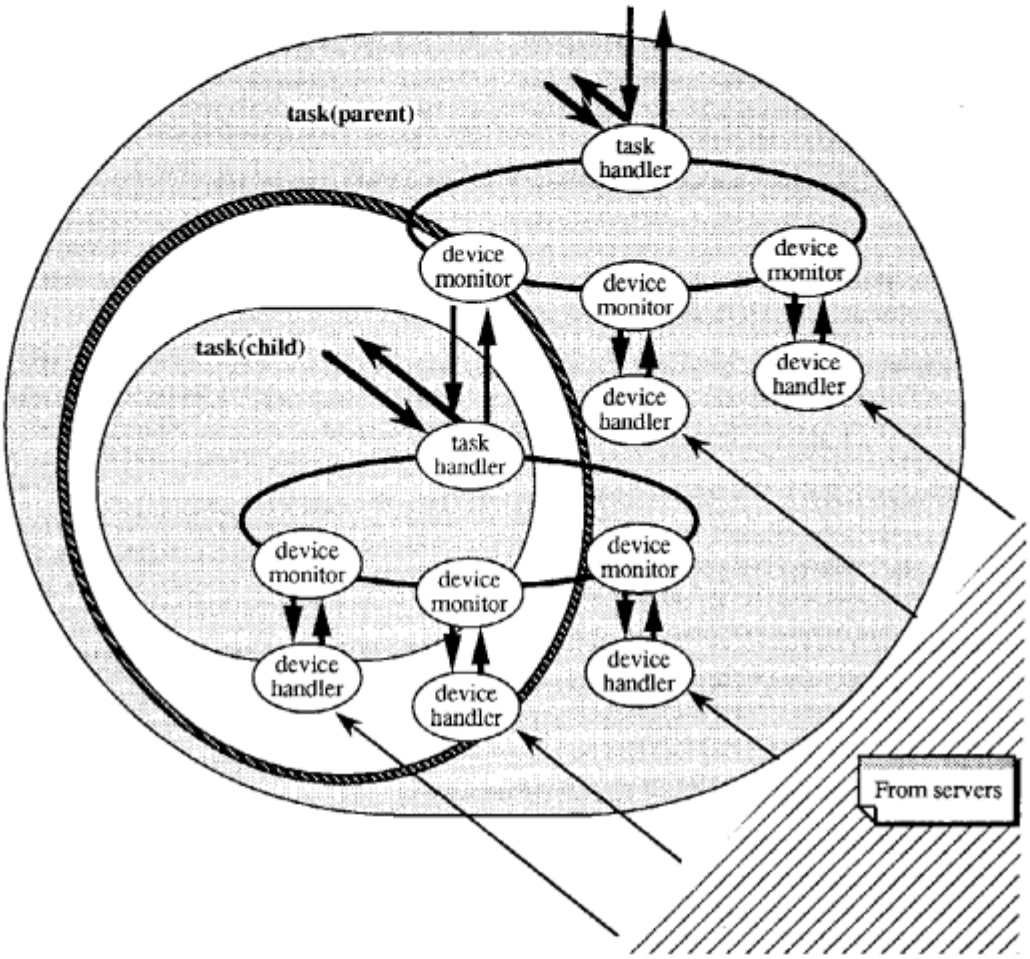


Figure 7: Resource tree

creator of the task (user programs) can only control and observe the behavior of tasks indirectly through requests to PIMOS.

General Request Handler: General request devices are the primary devices provided by PIMOS. Through them, information on the task itself is obtained and various other devices (including children tasks) can be created.

Standard I/O Handler: Standard I/O devices are aliases corresponding to some other device. They provide the resource sharing mechanism described above.

Server Device Handler: Server devices are the most common form of virtual devices provided by PIMOS. The device handlers watch the status of the client task and notify its termination to the server task.

3.2 Providing Services

To minimize the "kernel" of PIMOS, the kernel provides its fundamental resource management mechanism only. Other services, such as virtual devices such as files or windows, are provided by tasks called "servers".

Figure 8 shows an overview of the management hierarchy of PIMOS. The basic I/O system (BIOS) provides the low-level I/O, but it does not provide the protection mechanism. To protect the system, basic I/O service is provided only for the kernel. The kernel provides the above-described resource tree, which provides the resource management mechanism for tasks. Tasks here include both user program tasks and server tasks.

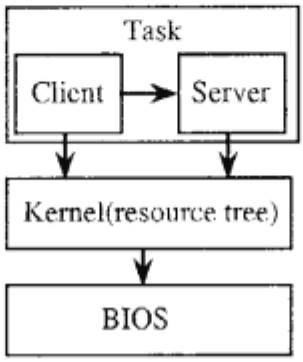


Figure 8: An overview of the management hierarchy

As described above, communication between the user programs and PIMOS can be established using the *raise* primitive. However, this mechanism only establishes a path to the kernel (the resource tree) and not to a server task.

The communication path between a client task and a server task can be established as follows (see Figure 9, also).

1. To start the service, servers register their service to the service table kept in the kernel of PIMOS. The table associates service names to a stream to the corresponding server. The code for the stream filter for protecting the server from clients' malfunction is also registered in the table.

2. The client task establishes a communication path to the PIMOS kernel and requests a service by its name.

3. The kernel searches for the name in the service table, and if a matching service is found, connects the client and the server, inserting a protection filter process inside the client.

Although the above written order is typical, The order of 1 and 2 is not essential. Requests made prior to registration of the service will simply be suspended.

In step 3, PIMOS inserts a device monitor and a device handler corresponding to the server device. The device handler watches for termination of the client task and notifies it to the server (Figure 10) for finalizing the service provided.

This separation of the kernel and the servers in PIMOS allows flexible configuration of the system and assures system robustness. Failures in a server will not be fatal to the

system; the services provided by the server will become unavailable, but the kernel of the system not to be affected.

Table 1 lists standard services in the most recent version of PIMOS (Version 3.2). Each of these services is implemented using the above client/server mechanism. Various other servers, such as database servers, can be added easily and canonically to these standard servers.

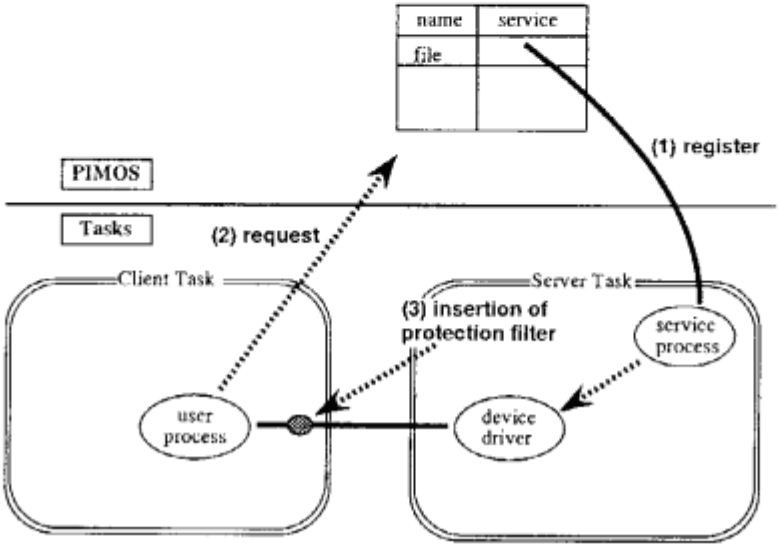


Figure 9: Communication between client and server(1)

Table 1: Standard service in PIMOS(Version 3.2)

Name	Service
atom	Database of atom identifiers and their unique printable names.
file	File and directory service.
module	Database of executable program codes.
socket	Internet socket service.
timer	Timer service.
user	Database of user authentication information.
window	Display window service.

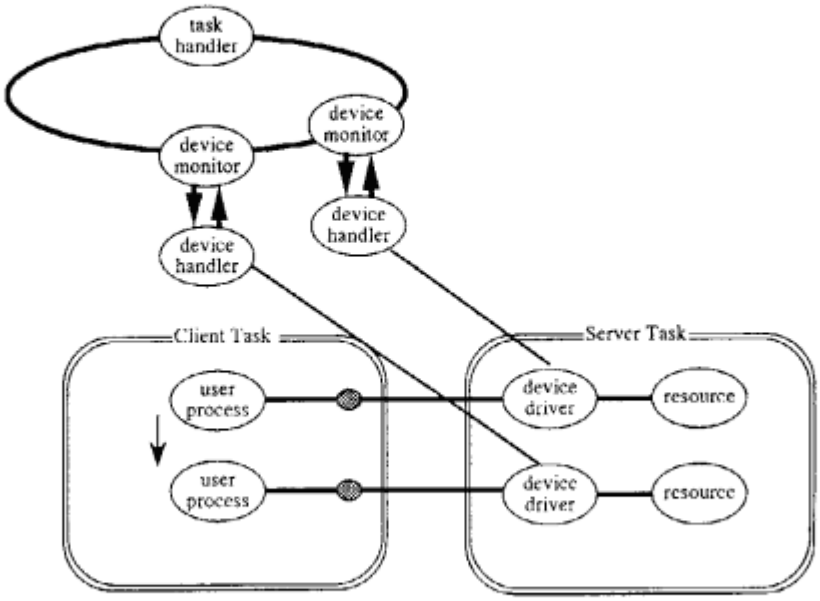


Figure 10: Communication between client and server(2)

3.3 Standard I/O

PIMOS provides a management mechanism for sharing resources, which enables the sharing of resource streams between a parent task and its children tasks (and subsequent children tasks). When a task is generated normally, standard I/O devices of the parent task are inherited to the child task. Multiplexing of the request bstream is implemented as described previously.

Standard I/O devices are not a usual device but a kind of alias of the device it is associated with. Since the protection mechanism of PIMOS, a messages filtering process, has to know the message protocol of the stream, the message protocol for the standard I/O device is restricted to a common subset of I/O device protocols.

3.4 Low Level I/O

In the lowest level, PIMOS supports SCSI (Small Computer Standard Interface) for device control. Each operation to the SCSI bus is provided as a built-in predicate by the KL1 language implementation. For example, a primitive for sending a device command through the SCSI bus is as follows.

```
scsi_command(SCSI, Unit, LUN, Command, Length, Direction,
 Data, DataP, ^NewData, ^TransferredLength, ^ID, ^Result, ^NewSCSI)
```

The argument `SCSI` should be an object representing the state of the SCSI bus interface device at a certain moment. `NewSCSI`, on the other hand, represents the state of the device *after* sending the command. This is instantiated only after completing the operation and the value will be used in the next operation, which will be suspended until it is instantiated. The proper ordering of operations is thus maintained.

The `Unit` and `LUN` arguments designate a specific device connected to the SCSI bus.

Arguments `Command` and `Direction` are used to control communication on the SCSI bus. The argument `ID` is used for command abortion, whose mechanism is similar to one described previously.

Since the KL1 processor needs garbage collection, real-time programming in KL1 is basically impossible. On the other hand, physical operations on SCSI require real-time response. The above primitive only reserves the operation and actual operation will be done eventually, with lower level real-time routines. Explicit buffers are used to synchronize the activities of their lower level routines with KL1 programs. Other arguments, `Data`, `DataP`, `NewData`, `TransferredLength` are used to specify such buffers.

3.5 Virtual Machine

As all the communication between the user programs and PIMOS is initiated through the control and report streams of the shoen which implements the user task, a user program can emulate PIMOS and make application programs run under its supervision. This is useful for debugging application programs.

The same technique can also be used to debug PIMOS itself by writing a BIOS emulator, as all the other parts of PIMOS communicate with BIOS through paths established using the shoen mechanism. Figure 11 depicts an actual implementation of a virtual machine on PIMOS. As the virtual machine is a usual task in PIMOS, the protection mechanism of PIMOS prevents failures in the version of PIMOS being debugged on the virtual machine from being propagated to the real PIMOS. This facility has been conveniently used in debugging the kernel of PIMOS.

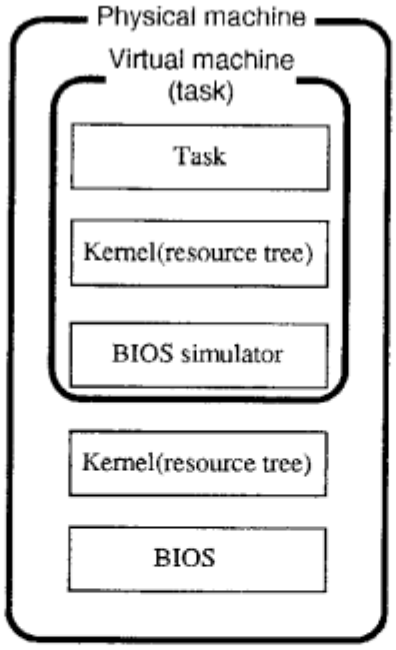


Figure 11: Virtual machine on PIMOS

4 Conclusion

The resource management scheme used in PIMOS based on the concurrent logic language KL1 is described. It depends heavily on the meta-level control mechanism called shoen provided by the language for efficient hierarchical resource management.

PIMOS itself has a hierarchical structure, consisting of a kernel and server tasks. This structure enables a flexible system configuration and reinforces the robustness of the system.

The system consisting of parallel inference machines (Multi-PSI and recently PIM) and earlier versions of PIMOS has been heavily used in research and development of experimental parallel application software for about three and a half years already, proving the feasibility and practicality of implementing an operating system in concurrent logic languages.

Acknowledgement

Many of researchers of ICOT and other related research groups. Too numerous to be listed here, participated in the design and implementation of the operating system itself and development tools. We would also like to express our thanks to Dr. S. Uchida, the manager of the research department of ICOT, and Dr. K. Fuchi, the director of the ICOT research center, for their valuable suggestions and encouragement.

References

[Chikayama *et al.* 1988] T. Chikayama, H. Sato and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988, pp.230–251.

[Foster 1988] I. Foster. Parlog as a Systems Programming Language. *Ph. D. Thesis*, Imperial College, London, 1988.

[Goto *et al.* 1988] A. Goto, M. Sato, K. Nakajima, K. Taki and A. Matsumoto. Overview of the Parallel Inference Machine Architecture (PIM). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988, pp.208–229.

[Nakajima *et al.* 1989] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa and T. Chikayama. Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proceedings of the Sixth International Conference on Logic Programming*, 1989, pp.436–451.

[Shapiro and Takeuchi 1983] E. Shapiro and A. Takeuchi. Object Oriented Programming in Concurrent Prolog. In *New Generation Computing*, Vol.1, No.1(1983), pp.25-48.

[Shapiro 1984] E. Shapiro. Systems Programming in Concurrent Prolog. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages*, 1984.

[Taki 1992] K. Taki. Parallel inference machine PIM. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, 1992.

[Ueda 1986] K. Ueda. Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. Technical Report TR-208, ICOT, 1986.