CEPBA

Centro Europeo de Paralelismo en Barcelona
Dpt. Arquitectura de Computadores
Universidad Politecnica de Cataluna

Paros Kernel Free Distribution v1.2 User Manual

Cristina Pujol, Jesus Labarta, Luis Gregoris, Sergi Girona, Toni Cortes

cris@ac.upc.es, jesus@ac.upc.es, luisg@ac.upc.es, sergi@ac.upc.es, toni@ac.upc.es

May 30, 1994

Work partially funded by the CEE under ESPRIT Project 2528 Supernode II "Operating Systems and programming Environments for Parallel Computers", by the Spanish Ministry of Education under grant TIC 392/89, and also under the grant APPARC 6634 BRA III.

Contents

Part I

Introduction

Chapter 1

What is this?

This manual accompanies the Paros Free Distribution v1.2 which is a version of the Paros Kernel freely available by anonymous ftp from *ftp.ac.upc.es*.

Paros is a microkernel for Distributed Memory Parallel Computers (DMPC) initially designed and implemented within the EC funded Esprit Program Project P2528 Supernode II. Supernode II started in March 89 and ended in December 92. Further work has been carried out from January 93 at the Computer Architecture Department of the Polytechnic University of Catalonia (UPC) and the European Center for Parallelism in Barcelona (CEPBA), under the support of grant TIC 392/89 from the Spanish Ministry, and also under the support of the grant APPARC 6634 BRA III.

The Paros microkernel has been designed to provide dynamic support for general purpose applications and operating systems under the distributed memory programming model. That means the simultaneous support of both a standard system interface and functionality (Unix, PCTE), which takes the advantage of the multiprocessor architecture to increase throughput, and a very efficient support for parallel programs willing to use several processors in order to reduce their execution time.

Paros has also been designed to allow several applications running concurrently by a single or several users. The need to support users with very different requirements from the functional and performance point of view leads to the specification of a low and very efficient *lightweight* kernel. This lets parallel applications to obtain a satisfactory performance of the machine. *Heavyweight* systems can be built on top of the *lightweight* functionalities, providing a smooth integration between parallelism support and system oriented services. In Paros Free Distribution v1.2, the Paros Kernel comes along with subsystems that support CSP and SMPD programming models; standard I/O services and also utilities for application building, monitoring and tuning. The environment constitutes a natural extension of UNIX like system support to parallel applications, freeing the programmer and user from the restrictions of a specific hardware platform.

The objective of the Free Distribution version is to let users outside the Computer Architecture Department use the kernel and provide feedback on the functionalities and performance for application development. Until now all the new features added to Paros Kernel have been validating the initial interface used to design Paros. Actually, Paros give support to different parallel applications, from traffic simulators to differential equations solvers. Several projects are being developed on top of Paros, covering different functional aspects. Some of those projects are:

- a debugger
- a cache for the file server
- a driver to access a disk connected to a transputer directly
- a driver for an ethernet board

The development of these projects are demonstrating the portability, hardware adaptability and reliability of the Paros Kernel.

Although this is not yet a commercial product, we are committed to support, through e-mail, as far as possible this Free Distribution. If you have any problem please mail to:

paros@ac.upc.es

Also mail to this address if you want to be included in a Paros mailing list and if you are interested in participation in open discussions about its functionality and features.

This manual is structured in the following way. We start with the release notes of this version and the previous ones, summarizing the improvements over version 1.0. This helps to keep up with the latest improvements, no matter what version you were running or if you just joined us as a new user. Then, we give a general description of what Paros is and how it relates to other systems within the area of Operating Systems and Programming environments for DMPCs.

Next, a more detailed description of the concepts and functionalities supported by Paros is given.

In chapter 5 to 6 we give an introductory session of simple examples showing the utilization and usefulness of the features supported by Paros. You may proceed directly to this session to get an overview of the present possibilities.

Chapters 7 to 9 are the manuals of the kernel interface of Paros Free Distribution, library functions, and utility commands that accompany it.

Appendix A describes the requirements and installation procedures. In this version we have also included a new appendix (appendix B) in order to give an example of how a sequential program can be parallelized on top of Paros, taking profit from the interface offered by this system. We conclude with a glossary of terms.

Partners involved in the Operating System work-package of the ESPRIT project were Thorn-Emi, SYSECA, Parsys, Telmat, IMAG, Aptor and UPC. Other partners in the project were Liverpool University, DRA, NAG, NA Software, DPC, Alsys, Ipsys, ARUP, Inmos and APD.

Chapter 2

Release notes

2.1 Release notes v1.2

Several improvements over Paros Free Distribution version 1.1 have been made at kernel, subsystem, libraries and application level.

This new version has been prepared to run not only on different host operating systems (DOS, Unix, IDRIS V) but also to be run as a multiuser operating system. In this case, several users can execute several applications in parallel. These users work on different clusters that protect one user from another. Anyway, there might be some interferences between them caused by through routing. Paros Multiuser is actually being tested on a Supernode with 26 transputers, hosted by a PC with SCO Unix.

Another new feature is the possibility to execute *make* on top of Paros. Makes can be executed sequentially or in parallel. In the case of a parallel make, it will try to do as many objectives in parallel as possible, depending upon the processors and memory availability. A small cache has also been implemented in order to increase the speed in compiling and linking steps.

The kernel has also been improved, some detected bugs have been checked and corrected, and parts of the kernel code have been optimized. In some system calls, frequently used, the number of messages between the calling process and servers have been reduced, obtaining better performance. The system calls related with thread suspension has been modified. When suspending all the threads of a task or Ptask, the status of those threads are returned to the calling process. No information is stored at the kernel level. Those threads can be resumed by the system call used to create threads *Thread_create*. For that reason the system calls *Ptask_resume* and *task_resume* have been removed from the Paros interface. This simple mechanism allows the programmers a great flexibility and at the same time lightens the kernel, allowing higher performance and more simplicity in thread managing.

New features has been added to the file server (*stdio_dup, stdio_cancel, remote_read*). These new features were added in order to offer a better support to other subsystem servers. The file server has also been prepared to serve the needs of a multiuser operating system.

The loader has been improved increasing the fault tolerance to configuration file errors. The loading speed has also been increased loading, in parallel, the code that will be executed by the different processes of a user application. Both, CSP and SPMD like program models are actually supported by this loader.

Some complex Built-in commands of the command line interpreter (*linkalos* and *cubata*) have been removed and are now Non-built-in commands, in order to simplify the interpreter execution flow. The use and results of these commands are exactly the same, but they are just not integrated with the command line interpreter, they are run as applications. This allows to keep small the command line interpreter, in order to be able to run our system in processors with 1Mbyte of external memory.

The graphic server has also been improved with a more efficient internal implementation.

Within this new version, the Host server is provided, replacing the afserver which was needed for the last version. This new server (**hostser**) has been re-written and modified in order to support makes, by adding new functions to it, and it has also been improved adapting some functions to Paros requirements. Some other functions have also been added to improve the graphic server performance. No other functionality of Paros Free Distribution v1.2 has been affected by the changes in the Host server.

Paros operating system has been designed to provide dynamic support for both application developers and system programmers. It is an important research and didactic tool on which a great amount of people are working on, from university to phd students. Actually several projects are being carried out on top of Paros in order to improve the system globally. Among that work, we will point out the followings: a distributed cache for the file server in order to improve the I/O calls performance; a disk server for a Supernode in order to avoid the bottle neck of communicating with the Host operating system, to access its disk server; a network server to incorporate TCP/IP on top of Paros; a debugger; some user commands like grep; a new command line interpreter with more features; Parlib library. The incorporation of these new features demonstrates the low cost to adapt Paros to new hardware platforms as well as to improve and add new utilities. These validates the interface of Paros and proofs the great flexibility and portability of this operating system.

The next version of Paros will have included these new features, that are being developed.

2.2 Release notes v1.1

Paros Free Distribution v1.1 includes a number of improvements over version 1.0 at the kernel, subsystem libraries, tools and application levels.

The kernel implementation has been improved both in functionality and robustness. From the functionality point of view, primitives of the interface specification that were not offered in version 1.0 are now available. This includes the ability to delete an executing program, the timeouts in the port communication primitives and the ability to allocate a specific processor.

Regarding robustness, some bugs in memory management functions have been corrected. A more dynamic management of the kernel data structures enables the kernel to run on networks where the system processors have 1Mb of memory (PFD v1.0 was statically dimensioned and required at least 2Mb in one of the system processors).

Several improvements have been made at the subsystem level. New commands have been added to the command line interpreter to allow development and execution on Paros without having to exit to the host operating system. This includes direct access to the host editor and the integration of the development tools on top of Paros. The kill command has also been added so that an application can be stopped without disturbing other running applications.

VGA graphics are available in this new version and the screen display has been improved. This benefits the monitoring tools which run faster with less interference to the monitorized applications.

The speed of the file server to access host files has been improved, although it is still slow because we use, for portability reasons, the stand alone afserver based routines, to read and write from the root transputer to the files in the host disk. The implementation of the loader of sequential applications has also been improved.

At the library level, a new version of the stdio and memory management functions have been ported. 3L Fortran run time library has been added. It is now possible to run both C and Fortran programs.

As already mentioned, several tools have been more tightly integrated or ported on top of Paros to improve the development environment. The Inmos C configuring tool (cubata) now runs on top of Paros. We also added binary compatibility for sequential programs with the stand alone tools. It is thus possible to run on top of Paros the 3L C and Fortran compilers. As side effect of this porting, it is possible to run several compilations concurrently thus achieving an important improvement over the native stand alone transputer environment.

The monitoring tools have been improved at the presentation level taking advance of the better graphics support.

An application to show how the kernel supports process migration managed by the user code has been included.

Paros scripts-files has also been improved allowing parameters in them. New scripts-files for separate compilation and linking are offered both for C and Fortran source files to execute them on top of the PC host OS. You will also find two script files for separate compilation and linking to run on top of Paros together with the migration example (see section 6.4 for details about scripts).

Functionalities that are intended for the next version (v1.2) are debugging support and extension of application performance tuning tools, Inmos C compilers and loaders, parallel makes, parallel loaders for certain frequent topologies (rings, grids) without requirement of a configuration file, SPMD loader and a programming model in the line of PARMACS or MPI.

Part II

Understanding Paros

Chapter 3

Paros is ...

Paros is an operating system micro-kernel specifically targeted to distributed memory parallel computers (DMPC). Paros is a general purpose kernel that efficiently supports message passing parallel applications.

Paros integrates in a coherent interface functionalities encountered in two communities not so closely integrated as should be desirable: operating system micro-kernel and heavy weight operating system developers on one side and parallel programmers and parallel programming environment developers on the other.

Before we describe some basic features of Paros we will describe the characteristics of other existing systems or environments to which it is related.

3.1 Micro-kernels

Paros provides for DMPC functionalities comparable to those offered by Chorus or Mach. These micro-kernels have originated and evolved in the area of distributed systems and shared memory parallel computers and in general have been basically influenced by classical heavy weight operating systems for sequential and networked machines. They have nevertheless been proposed for DMPCs and porting efforts have been undertaken.

The process model directly supported by these kernel has only two levels, being the upper one a single address space where all the lower level threads execute. The notion of several independent address spaces being part of the same parallel program is not supported.

The communication mechanism between independent address spaces was originally intended for OS oriented functionalities and communication between independent processes. It was not designed for efficient parallel programming within a single application (although it may have efficient implementations in a shared memory machine by avoiding data movements).

As a result, attempts to build on top of these process model and communication mechanisms a parallel programming model and environment for DMPC will always suffer from serious inefficiencies. Implementing typical heavy weight operating system for a DMPC on top of such kernels will also imply difficulties such as keeping track of the actual set of tasks that correspond to a single program (command). Inefficiencies and protection flaws are other possible effects.

These problems are apparent for example in Chorus where an actor (process) started under the Unix subsystem control can itself create other actors which escape the Unix control.

As conclusion, we should state that the basic parallel programming model efficiently supported by these micro-kernels is shared memory. The different threads within an address space could execute in parallel and communicate efficiently through the shared memory supported by the architecture. Implementing on these kernels the classical Unix environment for sequential or shared memory programs is the natural way (as has already been done).

A final comment is the importance given in these kernels and subsystems that accompany them to the executable code compatibility with Unix. This eases portability of existing Unix applications. This is in itself an indication that these systems are actually targeting sequential applications. The utilization made of using several CPUs if available is basically to increase throughput and share I/O resources. This argument of code compatibility is no longer valid when a new instruction set is targeted.

3.2 Distributed systems

Other frequently encountered operating system architecture for DMPC is one of a distributed operating system running on the parallel computer as if it where a network of computers. Idris and Helios are examples of this approach.

In each processor there is a full copy of a heavy weight operating system kernel. Different processors share I/O resources very much in the same way it would be done in a computer network.

Again these systems have the same problems mentioned for micro-kernels of not directly supporting the concept of single parallel program consisting of several address spaces.

Communication between parts of a single application is done by using standard heavy weight communication mechanisms such as sockets (Idris). As this will in many cases be very slow, other alternatives are usually offered such as dividing the DMPC into the distributed system hosting part and the raw processors used as attachment (see next chapter). This alternative has the advantage with respect to the single host case of allowing a great flexibility in the configuration (dimensioning) of the system at boot time.

Other alternative to overcome the efficiency problem in cases where the interface is not a standard Unix interface (Helios), is to provide a large set of functions covering a very wide scope of functionalities and overheads. The hierarchical classification of such functions and their interrelations and mutual effects are not very clear.

Algorithms used for certain functionalities have been taken from distributed systems (for example flooding in Helios to locate servers). It is our believe that DMPC are substantially different from networks of computers. This applies to the communication bandwidth available but also in that the system is a single computer, located in a single site and managed by a single person. As such, considerations such as independent management and availability of the nodes (for example maintenance shutdowns) that have to be dealt with in a distributed system are not important here. So, although similarities between the DMPC and distributed system can easily be found, the appropriate approaches should in general be different.

3.3 DMPC programming environments

The most frequent environment for programming parallel computers is one of a host processor running a standard operating system plus a parallel programming environment. This environment consisting of run time libraries to be linked with the application plus possibly a server to be run on the host offering a more or less restricted access to the functionalities of the host operating system. The programming environment may also include some resource allocation commands (typically servers or extensions to the host operating system) and configuration tools to adapt at compile time the logical structure of the application to the physical restriction of the DMPC attached to the host.

Modes of operation usually offered are single user, multiple user or batch. This depends on one side on the nature of the host operating system (MS-DOS, UNIX). It is also dependent on the level of support of the operating system to manage the DMPC as a sharable resource. When multiple users are supported, this is usually based on rather long term (session) allocation of processors (subcubes, domains) to users in a spatial and run to completion mode. Similarly occurs in batch mode, where the whole machine is usually allocated to a single application for its whole duration. No possibility of dynamic allocation of processors is offered to a program once started.

The processor management mechanisms are thus rather static, resulting in a low utilization of the machine. In this respect, operating systems for these machines seem to be in the same stage of evolution of memory management in the times of fixed or variable partitions.

The tight relation between logical structure of the application and physical structure of the hardware, that these programming environments usually enforce, leads in many cases to important restrictions in the actual programming model supported as well as the portability of the applications developed.

Although message passing is a global term under which all these systems can be embedded, there are actually a great variety of actual programming models. These can vary from uniform SPMD programs with asynchronous messages with processor identifiers as destination (N-cube, Express) to complex topologies of possibly different tasks with point to point connections (Inmos tool set). The actual synchronization semantics of the communication primitives varies a lot and it should be pointed that minor subtle differences in synchronization semantics may lead to important differences in behavior (deadlock for example).

As a conclusion of the previous observations, we can conclude that porting a program from one machine to other supporting a different environment may thus be cumbersome. On the other side, the amount of existing programs for some of these environments is starting to be important and, similarly to what happens with dusty deck Fortran codes, it will we necessary to be able to run them on future machines. This raises the need for a portable platform on to of which existing environments can be efficiently implemented, while also supporting new developments.

3.4 Paros

Key to Paros is the vision of the whole parallel computer as a single machine as opposed to a collection of sites. Within this machine processors are just another resource to be managed. Paros thus offers a very uniform view of the machine where functions can be used by any running program (or part of a parallel program) irrespective of the physical location of the program or the resource. The rights of the requesting program are the only restriction to this accessibility. In Paros it is possible to create and use objects from anywhere in the machine as long as one holds the right to do so. This includes creating remote processes, allocating remote memory, sending or receiving from a port irrespective of the processor where it was created, and so on.

This uniformity feature, that is not very frequent in todays distributed memory systems, is similar to what is offered in sequential machines, where the availability of the functions depends on the user identification and not on the physical page of memory from where the call is made. Examples of inhomogeneity in todays systems are unability to access the file system from any processor (many transputer programming environments), unability to create processes in other processors (Chorus, many DMPC environments), not uniqueness of names between different sites (Mach), allocation of resources can only be done by the host (many DMPC) or locally (micro-kernels),...

Dynamism is also a key feature of Paros. Any active program can allocate and deallocate resources (processors, memory) at any time during its execution.

The clear separation between logical and physical processor concepts, in the same way that logical and physical address spaces are differentiated in classical memory management, adds a great flexibility in resource management. Special interest has been put in the specification of the Paros interface to allow users worried with the logical to physical mapping to have control of it (if they have the appropriate rights). Reversely, the kernel may be allowed to map logical processors onto the available physical processors thus fully supporting immediate portability of applications between different machines running Paros, independent of the physical structure of the machine.

This latest feature allows for a very dynamic sharing of the processors in the machine including spatial separation between applications as well as time sharing between concurrent applications within each physical processor.

As any micro-kernel, Paros does not offer the classical heavy weight OS functionalities (file system, network access, medium and long term process control,...), but it provides support for implementing those functionalities as subsystems on top of the lightweight kernel.

As any micro-kernel, Paros does not offer a specific programming model and environment, but directly supports a flexible process and communication model on top of which many programming environments can be mapped. The Paros process and communication model is specially targeted to efficiently support programming environments and models commonly used in the domain of distributed memory multiprocessors.

The Paros process model directly supports the concept of parallel program consisting of several address spaces, as well as the concept of very lightweight threads within each of these address spaces. It thus avoids the process control problems mentioned for other micro-kernels. Threads directly support programming practices such as multi-threaded servers, excess parallelism, asynchronous communication and true shared memory parallelism if each node in the DMPC is itself a shared memory multiprocessor.

The Paros kernel has been first implemented on transputers. It is nevertheless a general purpose kernel portable to any distributed memory machine. Two basic features of Paros, lightweight threads and efficient communication, have very direct support on the transputer, but many other present architecture target these two features.

With Paros, we provide subsystems offering some heavy weight OS functionalities and support of two programming models. A basic subsystem offers file access from anywhere in the machine. The programming models supported by this subsystem are basically the Inmos C and 3L Fortran models, extended to avoid the need of a root processor (uniformity, all task have the same functionality available). Also included in the loader is the ability to pass parameters from the command line to all the execution tasks, thus avoiding typical initialization problems such as having to recompile or reconfigure the application and/or having to set up special paths and code to circulate the initial data. Control on the termination of the applications is maintained so that resources can be reused immediately even if the application does not explicitly free them before termination or it terminates abnormally. Several applications can be loaded concurrently. Binary compatibility with existing sequential programs is also available. The stand alone tools can thus be run on top of Paros.

Chapter 4

Paros concepts and functionalities

In this chapter we give a brief description of the basic concepts and functionalities supported by Paros. A few of these functionalities are not available in this cut down version. Refer to chapter 7 for a detailed description of what is supported.

The system has been structured as a low level kernel and subsystem servers on top of it. The kernel provides a reduced set of simple and well defined basic abstractions which can be used by the application and subsystem programmers. Subsystem servers offer the classical functionalities of standard operating systems. Many subsystems can be simultaneously run on a single DMPC.

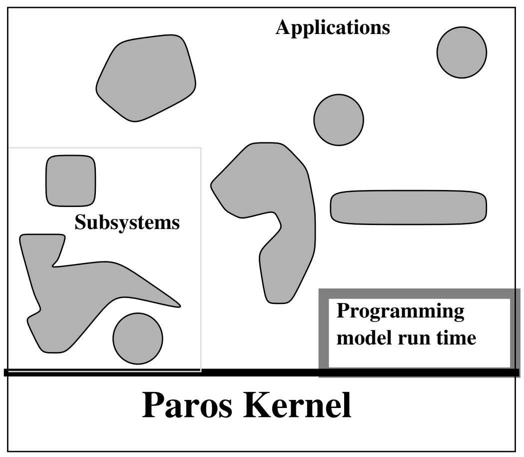

Figure 4.1:

Figure 4.1 shows how Paros offers a single system image at the microkernel level. Sequential or parallel subsystems programs, on top of it, provide the classical heavy weight operating system services such as file system, loaders,... User level applications have direct access to the kernel functionalities for performance reasons (the kernel provides mechanisms for the subsystems to keep control of the applications in this case). Although the Paros interface is far more dynamic and flexible than existing programming models, we provide, for portability reasons, the run time support required by other programming models.

The basic abstractions provided by the kernel can be grouped in process and communication model, resource management mechanisms (nodes and memory) and protection mechanism.

4.1 Protection model

The protection mechanism has two aspects: object protection and address space protection.

The operations on the global objects supported by the kernel that we will describe in a moment are protected through a **capability** based mechanism. Capabilities are partially encrypted.

The operations on objects that are local to a global object are not protected individually.

For each type of object we will mention the possible access rights. One general right is the owner right.

On creation of an object, a full capability is returned. A capability can be refined to produce new ones with less rights than the original. We provide a simple revocation mechanism based on total invalidation of previously distributed capabilities.

The problem of address space separation in a system based on processors with no hardware support (T800 transputers), poses many problems specially if the system has to be flexible and may have to support many concurrent programs. We provide a cluster concept as an address space protection domain. This concept also provides a mechanism for subsystems (UNIX, PCTE) to have control on the allocation of resources to parallel programs that will access directly the kernel interface without passing through the overhead of the subsystem.

The access rights on a **cluster capability** are: Owner, Cluster_resize, Ptask_create and Info.

A **kernel capability** is needed for some calls so that only very privileged processes can do these functions. The rights on such a capability may be: Owner and Cluster_create.

4.2 Process model

A three levels process model is directly supported by the kernel.

The **Ptask** concept encapsulates a single parallel program in execution. It

of tasks. A Ptask defines a local name space for its tasks, channels and resources like nodes and memory segments. All these objects within a Ptask define its logical structure. Besides creating and deleting Ptasks, the kernel allows the suspension of a Ptask which implies the suspension of all its tasks. To perform any operation on a Ptask, a **Ptask capability** is needed which may contain the rights : Owner, Ptask_Manage and Ptask_Segment.

A **task** is a logical address space in which several flows of control can execute. A task is mapped on one node within a Ptask, thereby all its threads will run on any processor in the node. The typical operations on tasks are also create, delete and suspend. A task suspension implies the suspension of all its threads.

A **thread** is a sequential flow of control within a task. All the threads in task share the same address space. Threads are very *lightweight*. In this respect they do not even have name so they may be created without intervention of the system (runp instructions can be used in a transputer based system without notifying the kernel). A thread can also destroy itself without notifying the kernel. This mechanism allows very fast thread creation and context switch. The transputers provide hardware support for this kind of threads.

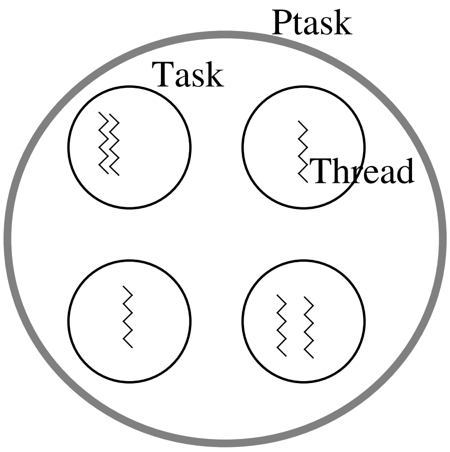

Figure 4.2: Process model

Threads are not initially provided for parallelism as they will run on a single node with possible a single processor and thus execute sequentially with an increased overhead over a purely sequential code. In DMPC where each node is itself a shared memory multiprocessor, threads could be used to achieve real parallel execution by multithreading a task. The use of several threads in a task might nevertheless be very useful to ease the implementation of servers (multi-threaded servers) as well as to easily implement asynchronous communications at the user level on a kernel that only provides synchronous communication mechanisms.

As threads are very lightweighted and do not have names, there is no way to stop or kill a single thread, unless it decides to stop by its own will. It is thus possible to kill *all* the threads in a task or Ptask by killing their containing task or Ptask. As in some applications, like debugging, it may be useful to suspend the threads of a parallel program, a suspend call is offered. For the reason stated above, only tasks and Ptasks can be suspended, but never a single thread. Suspending a task will suspend all the threads in it and return their state to the caller. As far as the kernel knows, suspending a thread is the same as killing it. In order to resume the threads, the user will have to create new threads with the state returned by the suspended call. This mechanism allows the programmers a great flexibility and at the same time lightens the kernel allowing more simplicity and higher performance in thread managing.

The kernel provides primitives for the creation of many tasks in a single call. Many threads on different tasks can also be created by a single kernel call.

We provide the basic support for exception processing in an integrated way with the communication mechanism. On Ptask creation an exception port (see section 4.3 for more information about ports) is specified. When the kernel detects an exception, a message is sent to the port with an indication of the type of exception and the state of the thread that caused it. The receiver of the message can implement the desired exception mechanism by using the memory and process management primitives. This mechanism provides the means to implement *heavyweight* operating system functionalities on top of Paros.

4.3 Communication model

The kernel supports two distributed memory communication mechanisms: ports and channels. The synchronization semantics in both of then is synchronous.

Ports are a global and flexible, n to n, protected, system oriented communication mechanism. They are targeted to allow communication between different Ptasks. It can be used to access subsystems and servers (sequential or parallel servers). Ports are identified by capabilities. A port capability may have the following rights: Owner, Read and Write. A simple name service mechanism for ports is also provided. Ports can be dynamically created and deleted.

Channels are a fast, 1 to 1, local communication mechanism within the tasks of a Ptask. It is the mechanism provided for parallelism, in duality with shared memory systems where the port mechanism also exists, but shared memory is the mechanism provided for parallelism. Channels have identifiers local to their Ptask and can not be used for communication between Ptasks. As they are local to Ptasks, overhead of access checks on global objects can be avoided. Channels can be dynamically created and deleted by the application in duality with the possibility to allocate and release memory in a shared memory system.

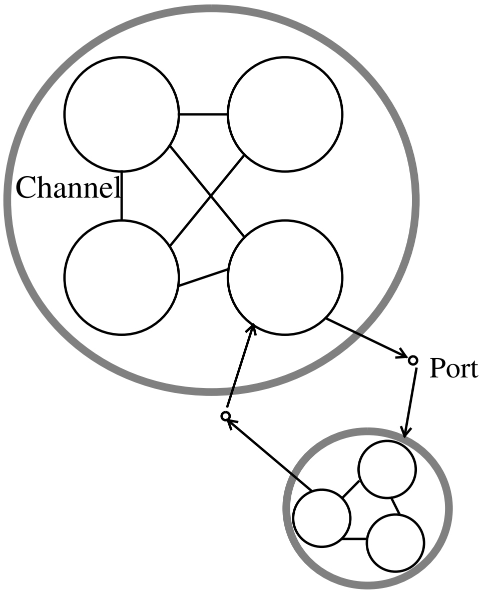

Figure 4.3: Communication model

For the kernel point of view, all communications mechanisms are synchronous with rendezvous because they avoid buffer maintenance and garbage collection problems on Ptask and task destruction. If an asynchronous communication is needed, it can be implemented with communication threads within a task.

The threads in a task are able to communicate through shared memory but the kernel does not specify shared memory synchronization mechanism. All synchronizations will have to be done by hardware mechanism (test & set, swap,...) or by software algorithms (Lamport, Eisenberg,...).

4.4 Processor management

We have differentiated between logical and physical nodes in much the same way that a classical operating system differentiates between physical and logical address spaces. The appropriate mechanisms and policies for managing nodes in a multiuser environment are not yet well understood. Many of todays users of DMPC want to manage the resources themselves . We think that kernel management of these resources can be very efficient (as is widely accepted for virtual memory in sequential nodes). Nevertheless the Paros interface allows the management policies (load balancing, migration,...) to be implemented at user level (subsystem) or automatically by the kernel.

Nodes can be dynamically requested by specifying a minimum acceptable number and a preferred number. A desired relation between logical and physical nodes can also be requested. This relation can be a one to one relation, which implies an exclusive use of the physical nodes, or an n to one relation allowing the kernel to map several logical nodes onto a single physical one. In order to support subsystem functions, such as installing drivers where the devices are physically located, it is possible to request the allocation of a specific physical node. It is also possible to request for nodes with a minimum memory space.

For those users confident on the system, only the logical structure of their application from the process model point of view needs to be specified. The same program without re-compilation will be mapped by the system on to the physical resources trying to achieve good performance. Users willing to have a closer control of the mapping can do it and take into account in their code the possible variations in resource availability. We have defined an interface that allows many different policies of node management to be implemented.

Process migration under the control of the user (migrate tasks between logical nodes) is supported. The kernel can also implement its management policies by controlling the mapping of logical to physical nodes.

As mentioned earlier, the **cluster** concept provides a mechanism for subsystems to control the global allocation of resources while letting application programs to access directly the kernel and specifically its node allocation interface.

The fact that Ptasks are created within a cluster also provides a mechanism to control the actual node time available to a Ptask. If only one Ptask is run in a cluster, all the physical nodes in the cluster will be fully devoted to that Ptask. If several Ptasks are created in a cluster they will all share the node time.

4.5 Memory management

A **segment** is a contiguous region of physical memory that can be allocated by specifying the logical node, size and base address if desired. Segments are allocated to Ptasks and thus only have local names. A segment can be mapped into the logical address space of at most one task. Of course, in the case of the T8, the logical address where a segment is mapped is the same as the physical address of the segment, but this will not be the case for other processors. A mechanism is provided to copy blocks of memory between segments to support subsystem functionalities like loading and I/O. This mechanism can also be used to support shared memory programming models although it was not originally intended for this purpose as reflected by the fact that source and destination Ptasks have to be specified which implies a series of protection issues.

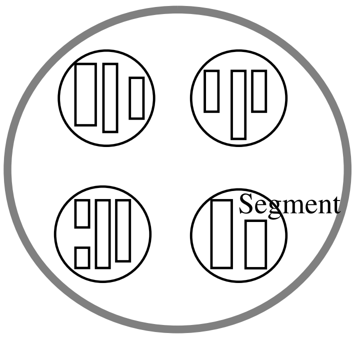

Figure 4.4: Memory management model

A Ptask may have segments allocated on nodes even if it has no task in that node.

Segments on any of the logical nodes of a Ptask can be allocated or released by any thread irrespective of where it is located. The only requirement is having a Segment right capability for the Ptask (threads within the Ptask can directly obtain such a capability from the kernel).

Part III

Tutorial

Chapter 5

Getting started

In this chapter we show some simple examples of program development on top of Paros Operating System. Paros Free Distribution v1.2 is basically the Paros micro-kernel. We provide some subsystems on top of it to allow the development and execution of Paros applications and support their runtime. The subsystems are: a command line interpreter with utilities such as edition, compilation and linking of applications, a file server to access files in the host system and a vga/cga display server in case of a PC hosted transputer network. We also provide some utilities for configuration and application building and monitoring, like make, cubata, mproc, limon,... that will be explained later.

The development environment for Paros Free Distribution consists thus of a host Operating System (MS-DOS, UNIX, IDRIS) with the 3L C and Fortran compilers and stand alone tools. Editing and application building can be done both in your host system with your favorite tools or on top of Paros without needing to leave the Paros environment (see section 9.2 to get complete information about the available tools).

To run Paros you need a transputer network with at least three processors. The first two processors are entirely devoted to micro-kernel and subsystem level management functions. The rest are available for user applications. It is recommended that you install the system before you go on reading, so you can try the examples as they are presented in this section (see appendix A for installation instructions).

Within this version we also provide more complex examples in order to illustrate with more details how to take advantage of Paros features (see appendix B).

5.1 Booting Paros

Currently Paros is booted by an application that runs on the Host Operating System. After booting Paros, this application acts as a passive server that provides access to the devices on the host, such as the screen or the host file system. All the kernel processes, subsystems and user applications of Paros Free Distribution run on the network of transputers.

To boot the Paros micro-kernel we provide a script that launches the host server and the micro-kernel files to be loaded on the transputer network. When you execute Paros:

```
c:PAROS>paros
```

The kernel will boot and a UNIX like command line interpreter is started (see section 9.1), which prompts:

```
Paros_?
```

You may now type commands (type *help* for a list of them). There is also a command (*shell*) to temporary exit to the Host Operating System.

5.2 Editing from Paros

It is possible to edit from the Paros environment by using the *edit* command, which calls your favourite editor (see section A.3 for installation instructions).

```
Paros_? edit hello.c
```

Let us now proceed to some simple examples.

5.3 Sequential hello world

Let us start by the classical hello world program in file *hello.c*.

```
#include <stdio.h>

main ()
{
   print ("hello world\n");
}
```

You will find this source code (and the following examples) within Paros Free Distribution v1.2 package, in the directory /paros/user/example.

5.3.1 How to generate executable files from Paros

Once you have created the file, you are ready to compile the application:

```
Paros_? comp hello
```

which will generate the file *hello.bin*. Then, you can link the application:

```
Paros_? link hello
```

which generates Paros executable file *hello.sn*.

Commands *comp* and *link* are both Paros script files (see section 6.4 for more details). The *link* command always expects an *indirect file*, with *.ind* extension. This *indirect file* is a text file that must contain a list of all the necessary object file names to generate the executable file. The *link* command expects this file even if there is only one object file. this type of files are also used by the 3L linker. In our case, *hello.ind* would only have:

```
hello.bin
```

You may write your own script files to generate an executable file. See also section 9.2 for more details about these tools.

5.3.2 How to generate executable files from the Host OS

You can also compile and link from the Host Operating System. All the explanations will be given considering DOS as the Host Operating System. First of all, you must exit Paros if it had been booted, by typing:

```
Paros_? exit
```

or

```
Paros_? quit
```

Now, you are ready to compile:

```
C:PAROS> ccomp hello
```

and link:

```
C:PAROS> clink hello
```

Both *ccomp* and *clink* are DOS script files (*.bat* files) that can be found in directory /paros/bin. *clink* command expects binary filenames (*.bin*) without extensions.

Once the executable has been generated, you can boot Paros again:

```
c:PAROS> paros
```

Compilation and linking steps can also be done by using the command *linkalos*:

```
Paros_? exit
C:PAROS> linkalos hello
c:PAROS> paros
```

linkalos expects also binary files (*.bin*) without extension. You can look and see how these DOS script files are constructed.

See also section 9.2 to get complete information about compiling and linking.

5.3.3 Running sequential applications

After compiling and linking you are ready to run this simple application. This program can be run as a sequential application by typing:

```
Paros_? hello.sn
```

which will execute it in a single processor and write into the screen the expected message:

```
hello world
```

In a UNIX like way you may redirect input or output.

```
Paros_? hello.sn >hello.out
Paros_? more hello.out

hello world
```

Also in a UNIX like mode, you may execute an application in **background**. Let us assume we have compiled and linked the file *mgs_s.c* (sequential version of the example that solves matrix ortonormalization with the Modified Gram-Schmidt method, see appendix B) into *mgs_s.sn*, found in directory /paros/user/matrix. We may execute:

```
Paros_? mgs_s.sn -m160 -n160 >mm1.out &
Paros_? mgs_s.sn -m160 -n160 >mm2.out &
Paros_? mgs_s.sn -m160 -n160 >mm3.out &
```

If your system has 3 processors available, the three concurrent applications will actually run in parallel, each on a different processor. If you have only one processor available, the three applications will execute concurrently within the same processor. You may now type:

```
Paros_? ps

Ptask status:
   pid   Time     Command
    3    00:08   mgs_s.sn
    4    00:05   mgs_s.sn
    5    00:03   mgs_s.sn
```

which will list the different applications concurrently running. The sequence shows how applications can be loaded in background and the status will be found in a UNIX like fashion.

The *ps* command can also be used to find out the actual mapping of logical to physical processors, as well as memory addresses of user segments, by only specifying the pid of the Ptask selected.

```
Paros_? ps 3

Ptask status:
  pid   Time   ppro   lpro   ntasks   nchans   seg   Command
   3    00:12    1      1       1        0       3    mgs_s.sn

  Physical processors list:     2
  Logical processors list:      0
  List of Tasks of Ptask:       0
  Channels list:
```

In this example the application was loaded in logical processor 0, mapped on to physical processor 2 (the identifiers of the physical processors start with number 2; the first two processors are reserved for the system). Note that the actual listing of the physical and logical processors shows the mapping from logical to physical processors. For that reason, you may see physical processor identifiers duplicated. The task identifiers are not listed in that way, because the mapping of each task on the corresponding logical processor is known by the user. They are listed in numerical order.

The processes identifiers (*pid*) that appears with *ps* command are unique within the system life.

See appendix B for a complet explanation about the previous example.

5.4 Parallel hello world

We are now interested in running a parallel program that prints the same message three times. Within Paros Free Distribution we provide, as a subsystem on top of Paros, a loader that supports CSP and SPMD like program models. Of course, the kernel provides support to implement on top of it loaders that provide other models.

5.4.1 CSP programming model

The CSP programming model drives the current loader with a configuration file. This file specifies the logical structure of an application (number of tasks and channels between them) as well as the resources it requires and the mapping of tasks to logical processors.

The configuration file format can be seen in the following example (*3tasks*):

```
Ptask description
number of processors :        3 logical
number of tasks :             3
number of channels :          0
number of executable files :  1

use /paros/user/example/hello.sn in 3 task
   T2 in logical processor 2
   T1 in logical processor 1
   T0 in logical processor 0

Task parameters
   T2 ( )
   T1 ( )
   T0 ( )
```

which specifies that our application will consist of 3 tasks, running the same executable file in this case. It also specifies that three processors are requested for this application and that each task will be mapped onto a different logical processor. The whole set of tasks will run as a single parallel program recognized as such by the kernel under the name *Ptask*.

An important feature of such a logical configuration is that it is analyzed at run time and the application is then mapped onto the available resources. This allows on one side very flexible and fine grain sharing, as resources are allocated and released on an application basis (not session). On the other side it provides portability of an application to sites with different physical sets of resources without having to modify the application structure or recompiling/reconfiguring it.

Generating Paros configuration files with cubata

Given that specifying such configurations for large applications may be cumbersome, within Paros Free Distribution v1.2 we also provide a tool that translates from Inmos C configuration language to the format accepted by the loader. The configuration file written in Inmos C configuration language for our previous example would be:

```
val m 3;            /* number of tasks */
val n m;            /* number of logical processors */

/*  Processor requirements */
processor m_processor[n];

/*  tasks description */
process hello[m];

/* tasks body and mapping to processor */
rep i = 0 to m-1
  {
    use "/paros/user/example/hello.sn" for hello[i];
    place hello[i] on m_processor[i];
  }
```

By running on top of Paros, the configuration utility *cubata* (ConfigUration BAsic TrAnslator) (gin + coke)

```
Paros_? cubata 3tasks.cfg 3tasks
```

we generate the configuration file *3tasks* that we have seen previously.

The Inmos C Configuration format is especially helpful when the communication between tasks uses a lot of channels. It is easier to resize your application with this format.

Next step would be to run the parallel hello application, which will be done by invoking the configuration file that you have just generated

```
Paros_? 3tasks
```

Which produces the following output:

```
hello world
hello world
hello world
```

The previous command loads a Ptasks with three tasks on three logical processors. This means that if your system has three available processors at the moment of running the application, each of the logical processors will be mapped to a different physical processor and correspondingly each task would be on a different processor. In the case that less than three processors are available, the kernel will group logical processors onto the available physical ones (and correspondingly their specified tasks will physically share CPU).

Had you been very sensitive to actually having three processors exclusively allocated to your application you could have specified *physical* instead of logical when requesting the processors in the configuration file.

Other important observation on the previous program is that any thread on any task has access to the file systems and UNIX standard library functions (*printf* in this case) irrespective of its actual mapping. Following the UNIX style, input and output can be redirected as for example:

```
Paros_? 3tasks >hello.res
```

which will redirect to file hello.res the output of the three tasks of the application. Running parallel applications with redirected standard input or output implies that ALL the tasks of a Ptask will shared the same redirected input/output. Heavy weight typical OS services, offered by subsystem processes and the objects managed by them, are global to a parallel program.

We may now proceed to modify the printed message in the following way:

```
#include "paros.h"
#include  <stdio.h>
```

```
main()
{
   int My_index;
   t_processor log_proc,ph_proc;

   get_task_index(&My_index);
   where_am_I(&ph_proc,&log_proc);
   printf("Hello world from task %d at logical processor
           %d\n \tmapped on physical processor %d\n",
          My_index,(int)log_proc,(int)ph_proc);
}
```

The output will certainly depend on the number of processors available but in case you have more than three in your system and you don't have other applications concurrently executing on top of Paros the output will be:

```
Hello world from task 0 at logical processor 0
      mapped on physical processor 2
Hello world from task 1 at logical processor 1
      mapped on physical processor 3
Hello world from task 2 at logical processor 2
      mapped on physical processor 4
```

or any permutation of these three messages.

The above example shows some new library functions. These are mechanisms by which a task in an application can identify itself dynamically within the logical structure of the application and the processor where it is mapped. This is done without having the user to program himself a clumsy initialization code that circulates this information, or having to specified it as parameter in the 3L configuration file which is a rather static solution.

The *get_task_index* function is supported by the loader and returns the number of the task in the configuration file at load time. It is not the real task identifier at the kernel level (the kernel allows for dynamic allocation of processors and creation of tasks by the application itself). The *where_am_I* function is trivially implemented on top of kernel level functions for obtaining the own identifiers and information. The source code of this function is given within Paros Free Distribution v1.2 package, to show an example of how to construct heavy weight functionalities on top of Paros.

5.4.2 SPMD programming model

An SPMD loader has been included in PFD v1.2. In this case a single executable file is loaded in several tasks, without setting up channels between them. All the tasks know their identifiers and have access to any Paros call that return information about the logical structure of the Ptask. It is thus possible for them to set up channels dynamically when communication is required.

The SPMD loader allocates a number of logical processors controlled by a header in the executable file. This header consists of two integers that give a lower and upper bound of the number of processors to be allocated. The real number of logical processors allocated will be between those bounds, but will depend upon the physical available nodes. There will be at least one logical processor mapped on each physical free or shared processor. The SPMD loader can not map any logical node on to physical nodes which were previously allocated with exclusion.

The header in the executable file is initialized by the linker. At link time, a program is interpreted as sequential, so these bounds are fixed at 1. There is a *patch* utility that can be used to modify those bounds when required.

A typical sequence for the *hello world* parallel example, after the linking phase, would be:

```
Paros_? patch hello.sn ?
Min = 1, and Max = 1
Paros_? patch hello.sn 3 3
Paros_?patch tse.sn ?
Min = 3, and Max = 3
Paros_? hello.sn
hello world
hello world
hello world
```

As you can see, when generating the executable file the bounds are set to 1 by the linker. After that, both bounds can be set to any desired value. In this example both values were set to three, so three tasks will be created and allocated in three different logical processors. If your system has three physical processors, those processes will be mapped on three different processors. All of them will execute the same code (hello.sn), producing the same effect as the example in the previous section (subsection 5.4.1).

In case the application needed to be loaded in the maximum number of physical processors (like the monitoring tools), a typical sequence to load the application would be:

```
Paros_? patch hello.sn ?
Min = 1, and Max = 1
Paros_? patch hello.sn 1
Paros_?patch tse.sn ?
Min = 1, and Max = -1
```

```
Paros_? hello.sn
hello world
hello world
hello world
```

Considering that our system had only three processor devoted to the users, the *hello world* would be loaded into those three processors, producing the same effect as the previous example. The free processors where the applications are loaded are **not** exclusively allocated. To allocate them exclusively, the CSP programming model must be used.

See also section 9.2.6 for more details about the *patch* utility.

Chapter 6

Carrying on

6.1 Parallel application

Let us now proceed to a simple parallel application with communication between the tasks (*distr.c*). The application estimates the mean value of a set of random values generated by a given function, the mean of the minimum of two such variables, the mean of the minimum of three, and so on.

/ file distr.c */*

```
#include "stdio.h"
#include "paros.h"

extern void randomize();
extern float random();
extern float average (float *, int)

main(int argc, char *argv[], char *env[], int n_chans,
      t_channelid *ch_list[])
{
   int   i, size, My_index, nb_tasks;
   float mean, *min, *buffer;
   int   length, error;

   get_task_index(&My_index);
   get_num_tasks(&nb_tasks);
   size=atoi(argv[1]);
   randomize();
   /* initialize heap*/
   malloc_init(2*size*sizeof(float) + 10*1024)
   buffer = (int *)malloc(size*sizeof(float))
```

```
   min = (int *)malloc(size*sizeof(float))
   /* generate private data */
   for (i=0;i<size;i++) buffer[i] = random();

   /* all but he first task must receive data from
    the previous one */
   length = size*sizeof(float);
   if (My_index != 0) {
      /* receive minimum from previous task */
      error = channel_in(*ch_list[0], (t_address)min, &length);

      /* minimum of the local and received values */
      for (i=0;i<size;i++)
         if (min[i]<buffer[i]) buffer[i]=min[i];
   }

   /* all but the last task must forward their result data */
   if (My_index < nb_tasks-1) {
     error = channel_out (*ch_list[1], (t_address)buffer,
                                        size*sizeof(float));
   }

   mean = average(buffer,size);

   printf("mean of minimum of %d random variables = %f\n",
                          My_index+1,mean);
}
```

The example shows how the standard *malloc* functions can be invoked by any thread to allocate memory locally (within its task, the heaps of tasks mapped in the same processor are different). The heap is actually implemented on a Paros segment allocated the first time a malloc function is called. The default heap size is 64K. If you foresee you will need more space you can create the heap by means of the *malloc_init* function with parameter the initial heap size before doing any allocation. It is important to note that execution of applications with lost pointers, or variables for which the *malloc* function failed, may cause problems to the system due to transputers lack of hardware translation for addresses.We recommend to control *malloc* errors and exit your application if detected.

A sample configuration file for this program could be named *distr* and contain the following logical structure:

```
Ptask description
number of processors :          5 logical
number of tasks :               5
number of channels :            5
```

```
number of executable files :    1

use /paros/user/example/distr.sn in 5 task
   T0 in logical processor 0
   T1 in logical processor 1
   T2 in logical processor 2
   T3 in logical processor 3
   T4 in logical processor 4

declare channels
   C0 from T0 to T1
   C1 from T1 to T2
   C2 from T2 to T3
   C3 from T3 to T4
   C4 from T0 to T0

Task parameters
   T0 ( C4 C0 )
   T1 ( C0 C1 )
   T2 ( C1 C2 )
   T3 ( C2 C3 )
   T4 ( C3 )
```

In this configuration, a linear chain of channels is formed. Each task does its random numbers list and then wait for the results from the previous one and after some computation sends its result to the next. The first task in the chain does not wait for results of the previous one nor the last one sends them out. After doing its average calculation each task prints its results. Communication and synchronization between contiguous tasks takes place through the channels that link them in the logical topology of the application. Channel C4 is only used for alignment of the input and output parameters to the tasks.

An equivalent logical structure can be generated by the *cubata* utility with configuration file *3tasks.cfg*

```
val m 5;           /* number of tasks */

/*  Processor requirements */
processor m_processor[m];

/*  tasks and channels description */
process (interface (input in, output out))hello[m];

/* tasks body and mapping to processor */
rep i = 0 to m-1
```

```
  {
    connect hello[i].out to hello[(i+1)%m].in;
    use "/paros/user/example/distr.sn" for hello[i];
    place hello[i] on m_processor[i];
  }
```

In this example you can also observe how channels are passed from the configuration file to the main function of each task. Although not shown in the example, several threads within a task can access the same channel, being the access serialized by the kernel.

The application can be built by using the command *linkalos*:

```
C:PAROS\EXAMPLES> linkalos distr rand average
```

It can also be built from Paros environment in the following way:

```
Paros_? comp distr
Paros_? comp rand
Paros_? comp average
Paros_? link distr
```

The command *link* will look for a file called *distr.ind*, which should contain the binary files that will be linked with Paros library \paros\bin\user.lib. The contents of *distr.ind* should be:

```
distr.bin
rand.bin
average.bin
```

But an easier and more efficient way to generate executable files is using *make files*. In this version of Paros, it is possible to construct make files and execute them on top of Paros. See section 6.2 and 9.2.4 for more details about *make*. Once the makefile has been written, you only need to type:

```
Paros_? make
```

and the executable file will be generated. You can also execute the parallel make by typing:

```
Paros_?make -j
```

The application can then be run:

```
Paros_? distr 100000 >distr.out& ps

Ptask status:
   pid   Time     Command
    1    00:01     distr

Paros_? ps 1

Ptask status:
  pid   Time    ppro   lpro   ntasks   nchans   seg   Command
   1    00:05    3      5       5        5       20    distr

  Physical processors list:     2 2 3 3 4
  Logical processors list:      0 1 2 3 4
  List of Tasks of Ptask:       0 1 2 3 4
  Channels list: C0(T0,T1) C1(T1,T2) C2(T2,T3) C3(T3,T4) C4(T0,T0)
Paros_? wait
Paros_? more distr.out
mean of minimum of 1 random variables = 0.500577
mean of minimum of 2 random variables = 0.333456
mean of minimum of 3 random variables = 0.250399
mean of minimum of 4 random variables = 0.199565
mean of minimum of 5 random variables = 0.166186
Paros_?
```

The *ps* command can be used to find out the actual mapping of logical to physical processors as well as the number of tasks and logical interconnection between them. Recall that the application may dynamically allocate or free resources as well as create or delete tasks and channels, and thus its logical structure at a given moment may be different from the one specified at load time. In the previous case some tasks finish their work before others. This can be observed if we keep doing *ps 1*.

In this example you can also observe how the Paros environment allows parameters passing to the user application.

The *wait* command will block the command line interpreter until all the applications, that are still running, finish their execution. This is very useful when monitoring applications and also in script files.

A further comment on the previous sequence of commands is that this is a cut down version of Paros aiming at being portable to a large number of common platforms. For this purpose we selected the afserver based stand alone tools. As a result some restrictions had to be introduced in the free distribution version compared to our first Paros implementation. One of the restrictions is due to the single user protocol in the afserver and the sequential nature of MS-DOS. The Paros Free Distribution command line interpreter runs on the root processor as well as the Paros Free Distribution file server. They share on a mutual exclusion basis the afserver communications. As a result if the command line interpreter is blocked waiting for input it actually blocks any other I/O activity of the running Paros applications. The objective of the *wait* command in the previous series is to block the command line interpreter so that application output can be displayed. Observe that if an application is loaded in the foreground, the command line interpreter will wait for its end and does not block I/O.

We may want to compare the performance of the previous application when the same number of tasks is loaded in less processors. We may use the following configuration file:

```
Ptask description
number of processors :          3 logical
number of tasks :               5
number of channels :            5
number of executable files :    1

use /paros/user/example/distr.sn in 5 task
   T0 in logical processor 0
   T1 in logical processor 1
   T2 in logical processor 2
   T3 in logical processor 1
   T4 in logical processor 2

declare channels
   C0 from T0 to T1
   C1 from T1 to T2
   C2 from T2 to T3
   C3 from T3 to T4
   C4 from T0 to T0

Task parameters
   T0 ( C4 C0 )
   T1 ( C0 C1 )
   T2 ( C1 C2 )
   T3 ( C2 C3 )
   T4 ( C3 )
```

where only the number of allocated processors and the mapping of tasks to logical processors has been modified, not the logical structure of the application.

We could also be interested in loading the first configuration while the second is still running. If we use the *ps* command we may now observe that the kernel has grouped logical processors of this configuration file into less physical processors. The kernel uses for the moment a simple allocation algorithm that tries to balance the processors load as well as reducing the interference between independent applications.

A final interesting run would be:

```
Paros_? distr 400000
```

with a configuration that either itself groups tasks in processors or requests more processors than available (and the kernel will do the grouping). We may obtain an output like this (the distr.c file in /paros/user/example actually checks for errors and prints them):

```
Heap initialization: error -16 in segment allocation
Can't get enough memory
Heap initialization: error -16 in segment allocation
Can't get enough memory
task_index 0: error -8 sending data to channel 10000
task_index 2: error -8 in channel_in from channel 10001,
                received length 1600000
task_index 3: error -37 in channel_in from channel 10002,
                received length 1600000
```

The important thing is to observe how an error propagates and the application ends up in a clean way without interfering with other possible applications or blocking the system. In this case tasks 1 and 3 where mapped to the same processors than tasks 0 and 2. As a result, their attempt to allocate the heap segment fails. The first two lines in the output are traces from the *malloc_init* routine. When those threads exit due to the error (exit status 1), the two tasks are deleted along with the channels they are connected to. Other threads reading or writing from/to those channels receive an error indication, that causes them to return and thus terminate the application. Applications can end by using *exit*, *exit_terminate* or with a *return* in the *main* of a program.

6.2 Make files

In Paros Free Distribution v1.2, it is now possible to maintain, update and regenerate the executable files using makefiles on top of Paros. *make* executes a list of commands associated with each target, typically to create or update a file of the same name. *makefiles* contain entries that describe how to bring a target up to date with respect to those on which it depends. The appearance of a typical makefile on top of Paros is equal to those on Unix, differing only in the lines where the compiler and the linker are specified:

```
CC = \paros\bin\afserver.sn -:o 1 -:b \tc2v1\tc.b4
CFLAGS = /I\paros\include /X /T8
LD = \paros\bin\afserver.sn -:b \tc2v1\linkt.b4
LD1 = \tc2v1\taskharn.t8
LD2 = \paros\lib\user.lib
PATCH = \paros\bin\afserver.sn -:b \paros\bin\pclinker.a4

distr.sn: distr.b4
  $(PATCH) distr.b4 distr.sn 4000

distr.b4: distr.bin rand.bin average.bin /paros/lib/user.lib
  $(LD) $(LD1)+$(LD2)+distr.bin+rand.bin+average.bin,distr.b4

distr.bin: distr.c
  $(CC) distr.c  $(CFLAGS)

rand.bin: rand.c
  $(CC) rand.c  $(CFLAGS)

average.bin: average.c
  $(CC) average.c $(CFLAGS)
```

In this example the main objetive (target) is the executable file *distr.sn*, which depends on the rest of the targets. Attention must be paid in the way to specify the location of compilers and linkers, as well as the parameters to be passed to them. It is important to take into account which is the Host operating system. The previous make file was prepared for Paros with DOS as Host operating system. See section 9.2.4 for more details.

To run the sequential make, you only need to type:

```
Paros_? make
```

and the objectives will be processed sequentially.

Due to the high number of compilations performed by users in each session, a parallelization of the Unix make has been implemented. When running the **parallel make**, it will try to do as many objectives in parallel as possible in different processors. To run the parallel make you only need to type:

```
Paros_? make -j
```

and the objectives being performed will appear in the screen.

```
        /paros/bin/afserver.sn -:o 1 -:b /tc2v1/tc.b4 distr.c
/I\paros\include /X /T8 (/paros/user/example)
```

```
        /paros/bin/afserver.sn -:o 1 -:b /tc2v1/tc.b4 rand.c
/I\paros\include /X /T8 (/paros/user/example)
        /paros/bin/afserver.sn -:o 1 -:b /tc2v1/tc.b4 average.c
/I\paros\include /X /T8 (/paros/user/example)
        /paros/bin/afserver.sn -:o 1 -:b /tc2v1/linkt.b4
\tc2v1\taskharn.t8+\paros\lib\user.lib+distr.bin+
rand.bin+average.bin,distr.b4 (/paros/user/example)
Linking code...
Code linked OK
        /paros/bin/afserver.sn -:b /paros/bin/pclinker.a4 distr.b4
distr.sn 4000 (/paros/user/example)
Time: 29624448
```

Note that make files are not Paros Script files. See section 9.2.4 for more details about *makefiles*.

6.3 Performance monitoring

Let us consider that by now our application runs correctly and we want to tune its performance. Within the Paros Free Distribution v1.2 you will find two monitoring applications that can be used to analyze the cpu load on the different processors. For the previous example one of them could be used in the following way:

```
Paros_? mproc -m500 -s30 -t4000 -l >load.res&
Paros_? distr 128
```

The application *mproc* consists of one task in each processor that by sampling the CPU queues reports the evolution of CPU consumption. A varied set of parameters can be passed to *mproc* tasks in order to adapt the granularity, overhead and duration of the measures (in the previous case we requested 500 mean values each of which obtained through 30 samples separated 4000 microseconds; this yields a rather small overhead). When the *mproc* application ends, the percentage of high and low priority (kernel and user level code respectively) CPU consumption are reported to the standard output. Results are for global CPU consumption and thus, if more than one application is loaded concurrently after the *mproc* application, the results will not differentiate the individual contributions to the load. The *mproc* application can also be directed to display the results graphically on the screen, and the possibility to choose the processors, whose monitoring information will be shown, is also available. A real time continuous monitoring application is also provided (*limon*). In the first case more detailed representation with less overhead is obtained. In the second a very large elapsed time can be observed, although on line graphics display may interfere with the application.

See section 9.3 for more details about *mproc* and *limon*.

6.4 Scripts

The command line interpreter accepts simple script files as input. A script file has the following appearance (file *mproc.sh*):

```
#paros
clear
echo
echo
echo Distribution of minimum of several random variables
echo
echo
distr 100000
echo
echo
echo CPU utilization for the previous application on physical
echo processors 0 and 1 will now be monitored
echo
echo starting monitoring application.....
..\load\mproc -w5 -m200 -p0 -p1&
echo starting user application.....
distr 100000
wait
```

It can be directly run by typing

```
Paros_? mproc.sh
```

The command first runs the *distr* application alone and then runs it again monitoring the load on processor 0 and 1. The trace of CPU consumption for processor 0 and 1 during the application run is displayed on the screen.

The command line interpreter requires script files to start by

```
#paros
```

as shown in the example. Script files admit all the commands offered by the command line interpreter (see section 9.1 for a description of the available commands), but it only allows one level of overlapping, as you can see in the following script file called *ex.sh*:

```
#paros
hello.sn
ex.sh
```

This is a recursive script file, that will first execute *hello.sn* and then call itself again, so it ends up calling infinite times the initial example *hello.sn*. Another functionality in scripts files are *parameters*, which can be passed to any script file. This can be observed in the script files prepared for compilation and linking steps on top of Paros:

File *comp* :

```
#paros
\paros\bin\afserver.sn -:o 1 -:b \tc2v1\tc.b4 $1.c
                    /I\paros\include /X /T8
```

The parameter in this script file is the file to compile. As you can see, parameters are indicated by the symbol **$n** and a number, which corresponds to the position in the parameter list.

You will find this script file in directory /paros/bin, together with other scripts to compile and link. You may prepare your own scripts to build an application. It is recommended anyway to perform compilation and linking steps using makefiles.

6.5 Programming in Fortran

If you are a Fortran programmer you can transport your applications or create new ones on top of Paros. Within this version of Paros, almost all the 3L Fortran library is supported (see section 8.5 for a more detailed description). Besides, you have the advantage of Paros features availability. Mathematical functions are also offered, but Fortran input/output is not supported. The input/output functions availables are those that exist for programs written in C language. To call those functions from Fortran programs, an interface must be done. To implement those interfaces the programmer must be aware that parameters are always passed by reference in fortran functions. We provide within this version, the interface for input/output calls and file handling (see directory */paros/user/fortran* for files *pinta.c* and *file.c*). The users of Fortran language may found this annoying, but the advantage is that, on top of Paros, I/O calls can be performed by any process, independently of their location.

To compile and link fortran applications, the user will find scripts to be executed on top of the PC host OS and on top of Paros, in the directory /paros/bin (see section 9.2 for more details about the available tools). We will now show how to build an application with some code written in fortran and some in C language. For that purpose we have prepared an example called *compercf* that will be found n directory /paros/user/fortran. The example measures the communication speed between two tasks through a pair of channels, but with the curiosity that one of the task was built from a C source file (*comperc.c*), and the other from a Fortran source file (*comperfo.f77*), thus performing the same program sequence. So in order to run the application both executable files must be built, by compiling and linking them separately. To generate the executable file comperf.sn the user should perform:

```
C:\PAROS\USER\FORTRAN> fcomp comperfo
```

then file pinta.c, which performs the I/O interface, should also be compiled

```
C:\PAROS\USER\FORTRAN> ccomp pinta
```

now you are ready to link the application by typing

```
C:\PAROS\USER\FORTRAN> flink comperfo pinta
```

flink expects binary files (*.bin*) without the extension.

You can also generate the executable file on top of Paros, in that case the sequence would be:

```
Paros_? compf comperfo
Paros_? compc pinta
Paros_? linkf comperfo
```

The command *linkf* expects an indirect file (*.ind*) containing the binary files that must be linked. In our example, *comperfo.ind* would contain:

```
comperfo.bin
pinta.bin
```

You can generate your own scripts or makefiles to run on top of Paros (see 6.4) or on top of the PC host OS. In any case, you must always remember to link with the library *fortran.lib* found in /Paros/lib directory.

The next step is to generate *comperc.sn*:

```
C:\PAROS\USER\FORTRAN> ccomp comperc
C:\PAROS\USER\FORTRAN> clink comperc pinta
```

or

```
Paros_? comp comperc
Paros_? link comperc
```

After building both executable files, you can execute the application as any other on top of Paros. For that purpose the following configuration file (*compercf*) has been constructed:

```
Ptask description
number of processors :         3 logical
number of tasks :              2
number of channels :           2
number of executable files :   2

use /paros/user/fortran/comperc.sn in 1 task
   T0 in logical processor 0
use /paros/user/fortran/comperfo.sn in 1 task
   T1 in logical processor 1

declare channels
   C0 from T0 to T1
   C1 from T1 to T0

Task parameters
   T0 ( C1 C0 )
   T1 ( C0 C1 )
```

This is an example of how different executable files can be loaded on different tasks, even communicating between them.

As you have observe, it is very tedious to construct this application. It is clear that make files would help to do it much easier. The make file for this application would have the following appearance :

```
CC = \paros\bin\afserver.sn -:o 1 -:b \tc2v1\tc.b4
CFLAGS = /I\paros\include /X /T8
FF = \paros\bin\afserver.sn -:o 1 -:b \tf2v1\tf.b4
FFLAGS = /I\paros\include /X /T8
LD = \paros\bin\afserver.sn -:o 1 -:b \tc2v1\linkt.b4
LD1 = \tc2v1\taskharn.t8
LD2 = \paros\lib\user.lib
LD3 = \paros\lib\fortran.lib
PATCH = \paros\bin\afserver.sn -:b \paros\bin\pclinker.a4

all:            comperfo.sn comperc.sn

comperfo.sn:    comperfo.b4
```

```
        $(PATCH) comperfo.b4 comperfo.sn 4000

comperfo.b4:    comperfo.bin pinta.bin /paros/lib/user.lib \
                /paros/lib/fortran.lib
        $(LD) $(LD1)+$(LD2)+$(LD3)+comperfo.bin+pinta.bin,comperfo.b4

comperc.sn:       comperc.b4
        $(PATCH) comperc.b4 comperc.sn 4000

comperc.b4:       comperc.bin \paros\lib\user.lib
        $(LD) $(LD1)+$(LD2)+comperc.bin,comperc.b4

pinta.bin:      pinta.c
        $(CC) pinta.c $(CFLAGS)

comperfo.bin:   comperfo.f77
        $(FF) comperfo.f77 $(FFLAGS)

comperc.bin:    comperc.c
        $(CC) comperc.c $(CFLAGS)
```

This makefile has been constructed supposing that the Host operating system is DOS.

6.6 More examples

Some additional applications are distributed both sources and some possible configurations. This includes some linear algebra code (tse) showing the use of multiple threads within each task to overlap communication and computation; an application that measures the communication speed between two tasks through a pair of channels (comperf); the same example as before but written in Fortran; a process migration example (migratio) and an example that simulates a network of queues (queue).

They are representative of the programming style and functionalities supported by the kernel and loader.

With them, some scripts are given to demonstrate some of the possibilities offered by the Paros kernel and the subsystems implemented on top of it.

You may execute the script file *sub.sh* in directory /paros/user/menu, which through a menu, allows you to run those examples (remember to check the configuration files of the examples to adapt them to your system number of processors).

The example provided to show how Fortran programs can run on top of PFD v1.2, measures the communication speed between two tasks through a pair of channels, but with the curiosity that one of the tasks was built from a C source file, and the other from a 3L Fortran source file, thus performing the same program sequence. The configuration file used show how to place different executable files in different tasks of the same Ptask.

Within this version, we also provide differents versions of an example that solves matrix normalization (see appendix B), in order to show some simple steps of how to take advantage of a system like Paros.

Part IV

Paros Reference Programmers Guide

Chapter 7

Paros Kernel interface

This section describes the Paros Kernel interface. It also indicates the features of that interface not supported in Paros Free Distribution v1.2.

7.1 intro

NAME

Paros.h

SYNOPSIS

```
/* Type of global objects */

#define T_KERNEL                    1
#define T_PROCESSOR                 2
#define T_CLUSTER                   3
#define T_PTASK                     4
#define T_PORT                      5

        /* rights on the kernel */

#define CLUSTER_CREATE_RIGHT      0x4
#define PROCESSOR_ALLOCATE_RIGHT  0x2


        /* rights on physical processors */

#define ALLOCATE_RIGHT            0x2

        /* rights on clusters */

#define PTASK_CREATE_RIGHT        0x4
#define CLUSTER_RESIZE_RIGHT      0x2

        /* rights on Ptasks */

#define PTASK_MANAGE_RIGHT        0x4
#define PTASK_SEGMENT_RIGHT       0x2

        /* rights on ports */

#define PORT_WRITE_RIGHT          0x4
#define PORT_READ_RIGHT           0x2

#define MAX_PORT_SERV              33
```

```
        /* common right (for all the capabilities) */

#define OWNER_RIGHT                 0x1
#define ALL_RIGHTS                  0xF

/* segments */

#define NO_CREATED                   -1

#define NULL_CAP               null_cap

#define INIT_NULL_CAP    {-1,{-1,-1,-1}}


/* fields on the flags parameter */

#define PHYSICAL                    0x1
#define PERSISTENT                  0x2
#define SYMMETRIC                   0x4
#define ONE_ON_ALL                  0x8

        /* Exception types */

#define EX_PTASK_CREATE               0
#define EX_TASK_CREATE                1
#define EX_PTASK_DELETE               2
#define EX_TASK_DELETE                3

#define OK      0

#define FALSE   0
#define TRUE    1


/* Error */

typedef int t_error;

/* type definitions */

typedef char   *t_address;
typedef long    t_bitstring[3];
typedef int     t_boolean;
typedef int     t_channelid;
typedef int     t_clusterid;
typedef int     t_logprocessorid;
typedef int     t_objectid;
```

```
typedef int     t_objecttype;
typedef int    *t_pdes;
typedef int     t_physprocessorid;
typedef int     t_portid;
typedef long    t_processor;
typedef int     t_Ptaskid;
typedef int     t_rightmask;
typedef int     t_segmentid;
typedef long    t_size;
typedef int     t_taskid;
typedef long    t_time;
typedef int    *t_wptr;


typedef struct    /* Capability */
{
   t_objectid   ident;
   t_bitstring  crypted; /* Crypted value of rights + key */
}t_cap;

typedef t_cap  t_clustercap;
typedef t_cap  t_kernelcap;
typedef t_cap  t_physprocessorcap;
typedef t_cap  t_portcap;
typedef t_cap  t_Ptaskcap;

typedef struct            /* Channel descriptor */
{
  t_channelid   channel_id;
  t_taskid      source;
  t_taskid      target;
} t_channel;

typedef struct            /* Thread descriptor */
{
  t_taskid      task_id;
  int           status;
  t_address     Iptr;
  t_address     Wptr;
} t_thread;

typedef struct            /* Segment descriptor */
{
  t_segmentid       ident;
  t_logprocessorid  logicalp;   /* logical within the Ptask */
  t_address         base;
  long              len;
```

```
  t_boolean          persistent; /* delete the physical segment on
                                     task deletion or not */
  t_boolean          on_chip;    /* NOT SUPPORTED */
} t_segment;

typedef struct            /* Task descriptor */
{
  t_taskid           ident;
  t_logprocessorid   tpt;      /* logical processor where it is mapped */
  t_segment         *seg_list; /* segments list to be mapped to the task */
  int                sl_size;
 } t_task;

typedef struct            /* Description of a Ptask */
 {
  /* number of physical processors */
  int                nb_physproc;

  /* logical processors list */
  t_logprocessorid  *logprocessor_list;
  int                lpl_size;
  t_physprocessorid *physproc_list; /* corresponding physical processor */

  /*  physical segments list */
  t_segment         *segment_list;
  int                sl_size;
.
  /* tasks list */
  t_taskid          *taskid_list;
  int                til_size;
.
  /* channel list */
  t_channel         *channel_list;
  int                chl_size;

 } t_Ptkinfo;

typedef struct        /* Description of a task */
 {
  /* logical processor where it is mapped */
  t_logprocessorid   logicalp;

  /* segments list mapped to the task */
  t_segment         *segment_list;
  int                sl_size;

  /* channels list accessible to the task */
```

```
  t_channel         *channel_list;
  int                chl_size;

 } t_tkinfo;

typedef struct
 {
   int               type;
   t_Ptaskcap        Ptask;  /* Ptask that caused the exception */
   t_taskid          task;   /* task that caused the exception */
 } t_excepmsg;


/* Paros Kernel Interface */

extern int cluster_create (t_kernelcap, int, t_clustercap *);
extern int cluster_resize (t_clustercap, int);
extern int cluster_delete (t_clustercap);

extern int Ptask_create  (t_clustercap, int *, int ,
                          int, t_portcap, t_Ptaskcap*);
extern int task_create   (t_Ptaskcap, t_task*, int);
extern int Thread_create (t_Ptaskcap, t_thread *, int);
extern int Ptask_delete  (t_Ptaskcap);
extern int task_delete   (t_Ptaskcap, t_taskid);
extern int Ptask_suspend (t_Ptaskcap, t_thread *, int *);
extern int task_suspend  (t_Ptaskcap, t_taskid, t_thread *, int *);
extern int get_Ptask_cap (t_Ptaskcap*);
extern int get_task_id   (t_taskid*);
extern int Ptask_info    (t_Ptaskcap, t_Ptkinfo*);
extern int task_info     (t_Ptaskcap, t_taskid, t_tkinfo*);

extern int allocate_processors (t_Ptaskcap, int *, int
                                int, t_logprocessorid *);
extern int allocate_physical_processor (t_Ptaskcap, t_processor,
                                        int, t_logprocessorid *);
extern int deallocate_processors (t_Ptaskcap, t_logprocessorid *, int);
extern int req_proc_with_mem (int mem_size, t_clustercap cluster_cap);

extern int channel_create (t_Ptaskcap, t_channel *, int );
extern int channel_delete (t_Ptaskcap, t_channelid);
extern int channel_in  (t_channelid, t_address, int *);
extern int channel_out (t_channelid, t_address, int);
extern int channel_alt (t_channelid *, int , int *);

extern int port_create    (t_portcap* );
extern int port_delete    (t_portcap  );
```

```
extern int port_send      (t_portcap, t_address, int, t_time);
extern int port_receive   (t_portcap, t_address, int *, t_time);
extern int port_select    (t_portcap *, int, t_time, t_portcap *);
extern int port_publish   (t_portcap, int);
extern int port_unpublish (t_portcap, int);
extern int port_lookup    (int, t_portcap*);

extern int allocate_segment (t_Ptaskcap, t_segment *, int);
extern int map_segment      (t_Ptaskcap, t_taskid,
                             t_segmentid, t_address *);
extern int copy_segment     (t_Ptaskcap, t_segmentid, long,
                             long, t_Ptaskcap, t_segmentid, long);
extern int which_segment    (t_address, t_segmentid *, long *);
extern int delete_segment   (t_Ptaskcap, t_segmentid);

extern int cap_refine (t_objecttype, t_cap, t_rightmask, t_cap*);
extern int new_key    (t_objecttype, t_cap, t_cap*);
```

NAME

error.h

SYNOPSIS

```
        /* Protection errors */

#define KEY_INVAL           -1
#define CNO_RIGHT           -2


        /* The identifier is not correct */

#define ERR_CLUSTERID       -3
#define ERR_PTASKID         -4
#define ERR_PORTID          -5
#define ERR_TASKID          -6
#define ERR_SEGMID          -7
#define ERR_CHANNELID       -8
#define ERR_LOGPROCID       -9
#define NO_IDFREE          -10
#define ERR_SIZECHANN      -11 /*number of channels exceed size of the
                                 channel table (per Ptask) */


        /* Some error has been detected at PE level */

#define ERR_SEND           -12
#define ERR_REC            -13
#define ERR_COPY           -14
#define ERR_STARTP         -15


        /* List errors */

#define LIST_OFLOW         -16
#define SOME_ERROR         -17


        /* Object errors */

#define NO_PROCFREE        -18
#define NO_LOGFREE         -19
#define NO_CHTFREE         -20
#define NO_ITEMFREE        -21
```

```
        /* Cluster management errors */

#define MORE_ALLOC         -22
#define NOT_EMPTY          -23

        /* Process information returned by task_delete call,
          it is not an error */

#define PTASK_DELETED       24

        /* Process & Processor management errors */

#define EXCEED_QUOTA       -25
#define SEGM_NO_MAPPED     -26
#define NO_PROC_AVAILABLE  -27
#define INCORRECT_BOUNDS   -28


        /* Memory management errors */

#define MORE_MAPPED        -29
#define ERR_LENGTH         -30
#define ERR_OFFSET         -31
#define NOT_ALLEXIST       -32
#define NO_MEMFREE         -33
#define ERR_MAPPED         -34
#define ERR_ADDR           -35
#define DIFF_LOGPROC       -36
#define ERR_WITHPROC       -37
#define SEGMENT_BUSY       -38

        /* Communication management errors */

#define EXCEED_REQUESTN    -39
#define EXCEED_MAXCOMW     -40
#define DELETING_PORT      -41
#define DELETING_CHANNEL   -42
#define ALREADY_BOUND      -43
#define NOT_BOUND          -44
#define ERR_SERV_NUM       -45


        /* Kernel implementation errors */

#define NON_LOCAL_PORT     -46
#define PORT_TIMEOUT       -47
```

```
        /* VGA SERVER  errors */

#define INVALID_MODE       -50
#define INVALID_CGA_MODE   -51
#define INVALID_VGA_MODE   -52

        /* not implemented kernel call */

#define INVALID_CALL      -100

        /* errno errors ( -(error+100)) */

#define ENOENT            -102
#define EACCESS           -113
#define EFAULT            -114
#define ENOTDIR           -120
#define ENAMETOOLONG      -178
```

7.2 cluster management

NAME

cluster_create

SYNOPSIS

```
#include <paros.h>

int cluster_create (t_kernelcap kernel_cap,
                    int quota,  t_clustercap * cluster_cap)
```

DESCRIPTION

Creates a new cluster returning in the structure pointed by **cluster_cap** a full capability for the cluster. The parameter **quota** gives the maximum number of physical processors the new cluster will be allowed to contain simultaneously in its life. Initially no processor is allocated to the cluster.

The kernel capability **kernel_cap** must have Cluster Create Right.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

Possible errors due to restrictions in the implementation are :

MORE_ALLOC The cluster table is full

NAME

cluster_resize

SYNOPSIS

```
#include <paros.h>

int cluster_resize (t_clustercap cluster_cap, int quota)
```

DESCRIPTION

Fixes to the value **quota** the new limit for the processor allocation allowed to the cluster identified by **cluster_cap**. If this value is smaller than the already allocated number of processors within the cluster, an error is returned. The cluster capability **cluster_cap** must have Cluster Resize right.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_CLUSTERID The identifier within the capability is invalid

Possible errors due to restrictions in the implementation are :

MORE_ALLOC The cluster has more processors allocated than the new quota value.

NAME

cluster_delete

SYNOPSIS

```
#include <paros.h>
```

```
int cluster_delete (t_clustercap cluster_cap)
```

DESCRIPTION

Deletes the cluster identified by **cluster_cap**.

The cluster must be empty (no Ptasks in it).

The cluster capability must have Owner right.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_CLUSTERID The identifier within the capability is invalid

NOT_EMPTY The cluster has Ptask(s)

7.3 process management

This section describes the interface functions related to process model and management.

NAME

Ptask_create

SYNOPSIS

```
#include <paros.h>

int Ptask_create (t_clustercap  cluster_cap,
                  int *nb_proc, int min_proc,
                  int flags, t_portcap  excp_port_cap,
                  t_Ptaskcap  *Ptask_cap)
```

DESCRIPTION

Creates a Ptask on the cluster specified by **cluster_cap** and allocates processors to the Ptask.

If the **flag** PHYSICAL is set, a minimum of **min_proc** physical processors have to be allocated (exclusive use) to the Ptask. An error will be reported if there are not enough free processors. Up to a maximum of **nb_proc** physical processors will be allocated if there are processors available. On return, the parameter **nb_proc** will indicate the number of logical processors created(one per physical processor allocated). The identifiers of the logical processors (to be used in successive calls) are from 0 to the return value of **nb_proc**-1.

If the **flag** PHYSICAL is not set, a minimum of **min_proc** logical processors have to be allocated (not exclusive use of the physical processors) to the Ptask. Up to a maximum of **nb_proc** logical processors are requested. The kernel is free to map the logical processors onto physical ones at will. On return, the parameter **nb_proc** will indicate the number of logical processors allocated. The identifiers of the logical processors are from 0 to the return value of **nb_proc**-1. The **flag** SYMMETRIC can be set in this case to indicate to the kernel the convenience of balanced mappings from logical to physical processors.

Allocation of physical processors is restricted to the limits imposed by the quota assigned to the cluster. This quota fixes the maximum number of physical processors that can be allocated to all the Ptasks within a cluster a any time (see *cluster_create*).

Note that processors are only allocated, but nothing is loaded or mapped on them.

The parameter **excp_port_cap** is a capability for a port to which the kernel will send a message for each exception that may be detected in any thread of the Ptask. If the parameter is a NULLCAP capability, no exception message will be sent.

The content of the message will be a structure of type **t_excpmsg** containing information on the Ptask and task that caused the exception as well as an exception type. Presently defined exceptions are:

EX_PTASK_CREATE The Ptask has been correctly created.

EX_TASK_CREATE A new task has been created within the Ptask.

EX_PTASK_DELETE The Ptask has been deleted.

EX_TASK_DELETE A task in the Ptask has been deleted.

These exceptions make possible for subsystems to keep trace of what an application is doing while the application can still directly call the kernel.

A full capability for the newly created Ptask is returned in the structure pointed to by **Ptask_cap**.

The cluster capability **cluster_cap** must have Ptask Create right.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_CLUSTERID The identifier within the capability is invalid

EXCEED_QUOTA The minimum number of processors exceeds the cluster quota

NO_PROCFREE There is not the minimum number of free physical processors

INCORRECT_BOUNDS *min_proc* bound is bigger than *nb_proc*.

Errors due to restrictions in the implementation :

MORE_ALLOC The Ptask array is full

NO_LOGFREE There is not the number of free logical processor entries

STATUS

Only exception EX_PTASK_DELETE is sent in the Paros Free Distribution Kernel v1.2.

NAME

task_create

SYNOPSIS

```
#include <paros.h>

int task_create (t_Ptaskcap Ptask_cap,
                 t_task *task_list, int tl_size)
```

DESCRIPTION

Creates a set of tasks local to the Ptask specified by **Ptask_cap** and initializes the structures that represent them.

task_list points to an array of size **tl_size** with one entry per task to be created.

For each task, the corresponding entry specifies on which logical processor (field **tpt**) of the Ptask it has to be created, and a list of segments (field **seg_list**) that have to be allocated and mapped into the task. Each segment descriptor (in the array pointed to by the task descriptor) will be mapped in the logical address space of the task at the address specified by the **base** field in the segment descriptor.

The task identifiers are selected by the kernel and returned in field **ident** of each entry. If a subset of tasks can not be created, the field **ident** in their corresponding entry will contain a negative value describing the error, but the other ones will be created. The same may occur with some segments within some tasks.

The Ptask capability **Ptask_cap** must have Ptask Manage right. A previous step must be to allocate the processors where the tasks will be mapped to.

When a task is created, no threads are started, only resources allocation is performed. The task will not start its execution until threads are created, with *Thread_create* or *runp* calls or other library calls like *lib_loc_thread_create or lib_loc_thread_detach.*

Note that it is necessary to fill correctly the following fields : the logical processor identifier (**tpt**) and the segment list size (**sl_size**), for each task, and the task list size parameter **tl_size**, before this call is executed.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_PTASKID The identifier within the capability is invalid

SOME_ERROR There is at least a task that was not created, the kind of error is specified at the **ident** field in the corresponding entry.

ERR_LOGPROCID The identifier of the logical processor for the task is incorrect.

NAME

Thread_create

SYNOPSIS

```
#include <paros.h>

int Thread_create (t_Ptaskcap Ptask_cap,
                   t_thread *thread_list, int tl_size)
```

DESCRIPTION

Creates or resumes a set of threads local to the Ptask specified by **Ptask_cap**.

thread_list points to an array of **tl_size** entries each of which represents a thread to be created/resumed.

Each thread is created/resumed in the task specified by the field **task_id** of its **t_thread** structure. If a subset of threads can not be created, no one of them is created and the field **status** of the incorrect threads will indicate the type of error.

The Ptask capability **Ptask_cap** must have Ptask Manage right.

Note that it's necessary to fill correctly the following fields : the task identifier (where the thread belongs to), the working (Wptr) and instruction (Iptr) pointers, for each thread, and the thread list size parameter **tl_size**, before this call is executed. An example of the use of this function can be found in the code of the file "fork.c", distributed with this version.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_PTASKID The identifier within the capability is invalid

SOME_ERROR There is at least a thread that can not be created, the kind of error is specified at its **status** field.

ERR_STARTP Some error has occurred at an internal implementation level

SEE ALSO

Ptask_suspend, task_suspend

NAME

Ptask_delete

SYNOPSIS

```
#include <paros.h>

int Ptask_delete (t_Ptaskcap Ptask_cap)
```

DESCRIPTION

Deletes all the tasks in the Ptask specified by **Ptask_cap**. This call releases all the resources that were allocated to the Ptask (segments, logical processors).

The Ptask capability **Ptask_cap** must have Manage right.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_PTASKID The identifier within the capability is invalid

NAME

task_delete

SYNOPSIS

```
#include <paros.h>

int task_delete (t_Ptaskcap Ptask_cap, t_taskid task_id)
```

DESCRIPTION

The task specified by **task_id** in Ptask identified by **Ptask_cap** is deleted. All the threads in the task are aborted (they will never be resumed again). If some thread was blocked on a communication request (either channel or port), the communication is aborted.

All the channels from or to the specified task are also deleted (a communication error will be reported to other tasks that were willing to communicate with this one through some channel).

The persistent segments mapped in this task are not deleted but unmapped. Non persistent segments are deleted.

If it is the last task of the Ptask, then the Ptask is also deleted (and their allocated resources are released).

Ptask_cap must have Ptask Manage right.

RETURN VALUE

Upon successful completion, the operation may return

OK If the task was not the last of the Ptask.

Or

PTASK_DELETED If the task was the last one in the Ptask. This is not an error, it indicates that the Ptask has also been deleted.

In case of failure, it returns one of the following values :

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_PTASKID The identifier within the capability is invalid

ERR_TASKID The task identifier is not valid

NAME

Ptask_suspend

SYNOPSIS

```
#include <paros.h>

int Ptask_suspend (t_Ptaskcap Ptask_cap,
                   t_thread *thread_list,
                   int *tl_size)
```

DESCRIPTION

All the threads of the Ptask identified by **Ptask_cap** are suspended, and the necessary information to resume those threads is returned in the list pointed by **thread_list**. This information includes the instruction pointer (**Iptr**), the work space pointer (**Wptr**) and the identifier of the task where the thread was mapped to (**taskid**). The kernel does not keep any information about the suspended threads.

If some thread is in the middle of a kernel call, the call will be finished before the thread is suspended.

If a thread is blocked in a communication request at the time the suspend call is invoked, the suspension will be delayed until the communication finishes in the case the synchronization point has been reached. In the case it has not been reached, the thread will be immediately suspended. The communication request will be issued again if the thread is later resumed.

Specific threads, tasks, or the whole Ptask can be later resumed by using the system call *Thread_create*.

Ptask_cap must have Ptask Manage right.

Parameter **thread_list** must point to an array of t_thread structures on which the state of the threads will be returned. The size of this array must be specified by **tl_size**. If the size of the list is less than needed, the call will return LIST_OFLOW and the necessary size will be returned by the **tl_size** parameter, in order to perform the call again without loosing the information of any suspended thread.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_PTASKID The identifier within the capability is invalid

LIST_OFLOW The space reserved for the thread_list was too small. The necessary array size for the thread_list is returned by the parameter **tl_size**.

SEE ALSO

thread_create

STATUS

This call is not offered in Paros Free Distribution v1.2.

NAME

task_suspend

SYNOPSIS

```
#include <paros.h>

int task_suspend (t_Ptaskcap Ptask_cap, t_taskid task_id
                  t_thread *thread_list, int *tl_size)
```

DESCRIPTION

Stops all the threads in the task **task_id** within the Ptask **Ptask_cap**. Information on the state of the suspended threads is returned in the list pointed by **thread_list**. This information includes the Instruction (**Iptr**) and work space (**Wptr**) pointers and the task identifier, to whom the thread was mapped to.The kernel does not keep any information about the suspended threads.

If some thread is in the middle of a kernel call, the call will be finished before the thread is suspended.

If a thread is blocked in a communication request at the time the suspend call is invoked, the suspension will be delayed until the communication finishes in the case the synchronization point has been reached. In the case it has not been reached, the thread will be immediately suspended. The communication request will be issued again when the thread is later resumed.

Specific threads or the whole task can be later resumed by using the system call *Thread_create*.

Ptask_cap must have Ptask Manage right.

Parameter **thread_list** must point to an array of t_thread structures on which the state of the threads will be returned. The size of this array must be specified by **tl_size**. If the size of the list is less than needed, the call will return LIST_OFLOW and the necessary size will be returned by the **tl_size** parameter, in order to perform the call again without loosing the information of any suspended thread.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_PTASKID The identifier within the capability is invalid

ERR_TASKID The task identifier is not valid

LIST_OFLOW The space reserved for the thread_list was too small. The necessary array size for the thread_list is returned by the parameter **tl_size**

SEE ALSO

thread_create

STATUS

This call is not offered in Paros Free Distribution v1.2.

NAME

stopp

SYNOPSIS

```
#include <paros.h>

void stopp ()
```

DESCRIPTION

Destroys the thread that makes the call.

No check is made to test whether the thread is the only thread of the task. The task is thus neither destroyed.

RETURN VALUE

None.

NAME

runp

SYNOPSIS

```
#include <paros.h>

void runp (t_address wdes, t_address iptr)
```

DESCRIPTION

This call creates a new thread in the task of the thread that invokes it.

RETURN VALUE

None.

NAME

Ptask_info

SYNOPSIS

```
#include <paros.h>

int Ptask_info (t_Ptaskcap Ptask_cap, t_Ptkinfo *Pt_info)
```

DESCRIPTION

Returns information about the structure and resources of the Ptask identified by **Ptask_cap**. This includes logical and physical processors, segments, tasks and channels.

Parameter **Pt_info** is a pointer to a structure that contains pointers to the arrays on which this information will be returned. It must also contain on invocation the maximum size reserved for those arrays. These fields will contain on return the actual size of the Ptask resources lists.

If the size reserved for a given array is less than the needed, only the provided entries will be filled, but in any case the real list size will be returned.

Note that it's necessary to fill correctly the size fields of the **Pt_info** : the logical processor list size (**lpl_size**), the segment list size (**sl_size**), the task list size (**tl_size**) and the channel list size (**chl_size**), before this call is executed.

Ptask_cap must have Ptask Manage right.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_PTASKID The identifier within the capability is invalid

SOME_ERROR There is at least one of the arrays for the information on the Ptask whose size was different from 0 and less than needed.

SEE ALSO

intro

NAME

task_info

SYNOPSIS

```
#include <paros.h>

int task_info (t_Ptaskcap Ptask_cap,
               t_taskid task_id,
               t_tkinfo *tk_info)
```

DESCRIPTION

Returns the information about the structure of the task **task_id** in Ptask **Ptask_cap**. This includes information such as the logical processor where it is, its logical address space (mapped segments) and the channels to which it has access.

Parameter **tk_info** points to a structure that contains pointers to arrays of channels and segments descriptors on which the information will be returned. On invocation, the integers **sl_size** and **chl_size** should contain the maximum size of the corresponding arrays. On return those fields contains the actual size of both lists. If the size reserved for a given array is less than the needed only the provided entries will be filled, but in any case the real list size will be returned.

The Ptask capability **Ptask_cap** must have Ptask Manage right.

Note that it's necessary to fill correctly the fields of the **tk_info** : the logical processor identifier (**logicalp**), the segment list size (**sl_size**), and the channel list size (**chl_size**), before this call is executed.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_PTASKID The identifier within the capability is invalid

ERR_TASKID The identifier of the task is invalid

SOME_ERROR There is at least one of the arrays for the information on the task whose size was different from 0 and less than needed.

NAME

get_Ptask_cap

SYNOPSIS

```
#include <paros.h>

int get_Ptask_cap (t_Ptaskcap *Ptask_cap)
```

DESCRIPTION

Returns in the structure pointed by **Ptask_cap** the caller Ptask capability with Manage and Segment rights.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

Errors due to restrictions in the implementation :

SEGM_NO_MAPPED The call has been made by a thread running outside of a mapped segment (lack of transputer support for address space protections).

NAME

get_task_id

SYNOPSIS

```
#include <paros.h>

int get_task_id (t_taskid *task_id)
```

DESCRIPTION

Returns the caller task identifier in the variable pointed by **task_id**.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

Errors due to restrictions in the implementation :

SEGM_NO_MAPPED The call has been made by a thread running outside of a mapped segment (lack of transputer support for address space protections).

7.4 processors management

This section presents the interface functions which offers a single system image of processors as resources with the separation between logical and physical processors.

Recall that functions in the cluster management section (section 7.2) set up limits to this resource availability, and some functions in the process management section also imply processor management functionalities (Ptask creation and deletion, Task creation and deletion,...).

Refer to the installation guide for physical processor identifiers binding to the hardware configuration.

NAME

allocate_processors

SYNOPSIS

```
#include <paros.h>

int allocate_processors (t_Ptaskcap  Ptask_cap, int *nb_proc,
                         int min_proc,  int flags,
                         t_logprocessorid  *logproc_list)
```

DESCRIPTION

Allocates a set of logical processors to the Ptask specified by **Ptask_cap**.

If the **flag** PHYSICAL is set, a minimum of **min_proc** physical processors have to be allocated (exclusive use) to the Ptask. An error will be reported if there are not enough free processors. Up to a maximum of **nb_proc** physical processors will be allocated if there are processors available. On return, the parameter **nb_proc** will indicate the number of logical processors created(one per physical processor allocated). The identifiers of the logical processors (to be used in successive calls) are returned in the array pointed by **logproc_list**. The Kernel allocates the set of smallest non used logical processor identifiers.

If the **flag** PHYSICAL is not set, a minimum of **min_proc** logical processors have to be allocated (not exclusive use of the physical processors) to the Ptask. Up to a maximum of **nb_proc** logical processors are requested. The kernel is free to map the logical processors onto physical ones at will. On return, the parameter **nb_proc** will indicate the number of logical processors created. The identifiers of the logical processors are returned in the array pointed by **logproc_list**. The Kernel allocates the set of smallest non used logical processor identifiers. The **flag** SYMMETRIC can be set in this case to indicate to the kernel the convenience of balanced mappings from logical to physical processors.

Allocation of physical processors is restricted to the limits imposed by the quota assigned to the cluster. This quota fixes the maximum number of physical processors that can be allocated to all the Ptasks within a cluster a any time (see *cluster_create*).

The Ptask capability **Ptask_cap** must have Ptask Manage right.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_PTASKID The identifier within the capability is invalid

EXCEED_QUOTA The minimum number of required processors exceeds the cluster quota

NO_PROCFREE There is not the minimum number of free physical processors

Possible errors due to the internals of the implementation are :

NO_LOGFREE There is not the necessary number of free logical processors

INCORRECT_BOUNDS *min_proc* bound is bigger than *nb_proc*.

SEE ALSO

cluster_create, Ptask_create.

NAME

deallocate_processors

SYNOPSIS

```
#include <paros.h>

int deallocate_processors (t_Ptaskcap Ptask_cap,
                           t_logprocessorid *logproc_list,
                           int lpl_size)
```

DESCRIPTION

Deallocates from the Ptask identified by **Ptask_cap** the logical processors whose identifiers are passed in the array pointed to by **logproc_list**. The number of processors that is deallocated is given by **lpl_size**.

In case there are tasks or segments mapped on to one of the logical processor that are going to be deallocated, it is not deallocated and an error is returned.

In the case all the logical processors of the Ptask are deallocated the Ptask is also deleted.

The Ptask capability must have Ptask Manage right.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_PTASKID The identifier within the capability is invalid

NOT_ALLEXIST Some of the logical processors do not exist. Nevertheless the existing logical processors have been deleted

NAME

allocate_physical_processor

SYNOPSIS

```
#include <paros.h>

int allocate_physical_processor (t_Ptaskcap Ptask_cap,
                                 t_processor processor,
                                 int flags,
                                 t_logprocessorid *log_proc)
```

DESCRIPTION

The physical processor with identifier **processor** is allocated to the Ptask identified by **Ptask_cap**. If the **flag** PHYSICAL is set, the processor selected should be free, if not an error will be reported. If the **flag** PHYSICAL is not set, the processor should be allocated without exclusive usage under the same cluster of the Ptask that requests it.

The logical processor number is returned in the variable pointed by **log_proc**.

The physical processor can be later deallocated using its logical name.

Ptask_cap must have Ptask Manage right.

See installation guide (section A.3) for a description of the mapping of physical processor numbers to the processors in your network.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_PTASKID The identifier within the capability is invalid

NOT_PROCFREE The processor can not be allocated

Possible errors due to internal implementation are:

NOT_LOGFREE There are no free logical processors.

NOT_CHFREE There are no free structures for channels in that processor.

NAME

req_proc_with_mem

SYNOPSIS

```
#include <paros.h>

int req_proc_with_mem (int mem_size, t_clustercap cluster_cap)
```

DESCRIPTION

Returns, if available, the identifier of a physical processor which has the amount of free memory requested (**mem_size**). This call does not allocate the processor; this step must be done by the system call *allocate_physical_processor*. The existence of this call is due to the process scheduling algorithm. When mapping the processes to processors, the kernel tries to balance the load among the nodes, but it has no way to know the applications memory needs when running. An application can be loaded on to a processor with enough memory to run it, but without the necessary memory that will be allocated at run time.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_CLUSTERID The identifier within the capability is invalid

NO_PROC_AVAILABLE There is no processor with the required free memory.

NO_PROCFREE There are no free processors within the cluster.

NAME

processor_info

SYNOPSIS

```
#include <paros.h>

int processor_info (t_cluster_cap cluster_cap,
                    t_processor physproc,
                    t_pp_info * pp_info)
```

DESCRIPTION

Returns in the structure pointed by **pp_info** information about the physical processor of identifier **physproc**. This information includes the percentage of free memory, the largest segment size, the number of logical processors mapped into it and the identifier of the cluster the processor belongs to (this field will have NO_CLUSTER value if the processor belongs to the pool of free processors).

The processor must be allocated to one Ptask within the cluster identified by **cluster_cap**.

The **cluster_cap** must have Info Right

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_CLUSTERID The identifier within the capability is invalid

WRONG_PROC The processor does not exist or does not belong to any Ptask in the cluster

STATUS

This function is not available in Paros Free Distribution v1.2.

7.5 communication management

This section describes the functions to manage and access communication objects.

NAME

channel_create

SYNOPSIS

```
#include <paros.h>

int channel_create (t_Ptaskcap Ptask_cap,
                    t_channel * channel_list,
                    int chl_size)
```

DESCRIPTION

A set of **chl_size** channels are created between tasks in Ptask **Ptask_cap**. Each entry in the array pointed to by **channel_list** corresponds to one channel, for which it specifies the task that will send to it and the one that will receive from it. This call returns for each entry the channel identifier that has been selected by the kernel.

If a subset of channels can not be created, their field *channel_id* will contain a negative value indicate why it could not be done, but the other ones will be created.

Ptask_cap must have Ptask Manage right.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_PTASKID The identifier within the capability is invalid

SOME_ERROR There is at least a channel that was not created, the kind of error is specified at its **channel_id** field.

NAME

channel_delete

SYNOPSIS

```
#include <paros.h>

int channel_delete (t_Ptaskcap Ptask_cap,
                    t_channelid channel_id)
```

DESCRIPTION

Is used to delete the channel of identifier **channel_id** within the Ptask specified by **Ptask_cap**.

In the case there is some thread willing to communicate through the channel, the communication will be aborted and an error reported to the affected threads.

Ptask_cap capability must have Ptask Manage right.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_PTASKID The identifier within the capability is invalid

ERR_CHANNELID The channel identifier is invalid

SEE ALSO

channel_in, channel_out, channel_alt, task_delete

NAME

channel_in

SYNOPSIS

```
#include <paros.h>

int channel_in (t_channelid chan_id,
                t_address address, int * size)
```

DESCRIPTION

The calling thread is blocked until a message arrives from the channel identified by **chan_id**. The incoming message will be stored in the buffer pointed to by **address**. The message will be truncated if the actual length of the message is larger than the value pointed by **size**. The actual length of the received message is returned in the variable pointed by **size**.

Several threads in a task may concurrently use a channel. The kernel guarantees mutual exclusion serializing the accesses.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

CNO_RIGHT There is no right for reading (IN_RIGHT) from this channel.

ERR_CHANNELID The channel identifier is not valid.

DELETING_CHANNEL The channel was deleted before the communication took place.

SEE ALSO

channel_out, channel_delete

NAME

channel_out

SYNOPSIS

```
#include <paros.h>

int channel_out (t_channelid chan_id,
                 t_address address, int size)
```

DESCRIPTION

Writes the array of **size** bytes pointed by **address** to the channel **chan_id**.

The calling thread is blocked until a thread in a task at the other end of the channel **chan_id** accepts the message.

The message will be truncated if the total buffer size allocated by the matching receive call is smaller than the total size specified to the send call.

Several threads in a task may concurrently use a channel. The kernel guarantees mutual exclusion serializing the accesses.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

ERR_CHANNELID The channel identifier is not valid for this task for output

DELETING_CHANNEL The channel was deleted before the communication could happen

SEE ALSO

channel_in, channel_delete

NAME

channel_alt

SYNOPSIS

```
#include <paros.h>

int channel_alt (t_channelid *chanid_list,
                 int chl_size,
                 int *channel)
```

DESCRIPTION

The **channel_alt** operation waits for a message pending on at least one channel among the array of **chl_size** channels pointed by **chanid_list**.

On return, the index **channel** indicates the position in **chanid_list** of a channel ready for input.

No guarantee is made that the channel will still be ready by the time a **channel_in** is made after a **channel_alt** if other concurrent threads try to read from the channels in **chanid_list**.(No mutual exclusion on the channels is assigned)

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

ERR_CHANNELID At least one of the channel identifiers is not valid for this task for input

DELETING_CHANNEL The channel was deleted.

NAME

port_create

SYNOPSIS

```
#include <paros.h>

int port_create (t_portcap *port_cap)
```

DESCRIPTION

A port is created. The call returns a capability with full rights for the new port in the structure pointed by **port_cap**.

All the processes that have the capability with the necessary right can read or write through this port (irrespective of their location).

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

Errors due to restrictions in the implementation:

MORE_ALLOC The internal port table is full

NAME

port_delete

SYNOPSIS

```
#include <paros.h>

int port_delete (t_portcap port_cap)
```

DESCRIPTION

Deletes the port identified by the capability **port_cap**.

If there are pending communication requests on the port, they will be aborted with a DELETING_PORT error indication.

The port capability must have Owner right.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_PORTID The identifier within the capability is invalid

NAME

port_send

SYNOPSIS

```
#include <paros.h>

int port_send (t_portcap port_cap,
               void *address,
               unsigned long size,
               t_time timeout)
```

DESCRIPTION

The calling thread is blocked until a receiver accepts the message at the port identified by the capability **port_cap**, or the **timeout** expires. The message of size **size** to be sent is stored at location **address**.

The message will be truncated if the total buffer size allocated by the matching receive call is smaller than the total size specified to the send call.

The port capability must have Write right.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_PORTID The identifier within the capability is invalid

PORT_TIMEOUT The timeout has elapsed

DELETING_PORT The port has been deleted while waiting for the matching receive call.

Errors due to restrictions in the implementation:

EXCEED_MAXCOMW This request exceeds the maximum number of allowed communications waiting for a message in this port

RESOURCES_EXCEED The kernel resources to implement the call are not available.

NAME

port_receive

SYNOPSIS

```
#include <paros.h>

int port_receive (t_portcap port_cap, void *buff,
                  int *size, t_time timeout)
```

DESCRIPTION

The calling thread is blocked until a message arrives at the port identified by the capability **port_cap**. The incoming message will be stored in the buffer pointed by **address** of maximum length pointed by **size**.

On return, the actually received length will be indicated in the position pointed by **size**.

If the message is larger than the total buffer size it will be truncated.

In case no *port_send* matches this request within **timeout** seconds the call will return an error.

The port capability must have Read right.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_PORTID The identifier within the capability is invalid

PORT_TIMEOUT The timeout has elapsed

DELETING_PORT The port has been deleted while waiting for the matching send call.

Errors due to restrictions in the implementation:

EXCEED_MAXCOMW This request exceeds the maximum number of allowed communications waiting for a message in this port

RESOURCES_EXCEED The kernel resources to implement the call are not available.

NAME

port_select

SYNOPSIS

```
#include <paros.h>

int port_select (t_portcap *port_select_list, int psl_size,
                 t_time timeout, t_portcap *port )
```

DESCRIPTION

The **port_select** operation waits for a message pending on at least one port among the array of **psl_size** ports pointed by **port_select_list**.

On return, the position pointed by **port** indicates a port ready for reception.

In the case the **timeout** elapses without any port in the list being ready, the call returns an error.

No guarantee is made that the port will still be ready by the time a **port_receive** is made after a **port_select** if other concurrent processes have Read capabilities for the selected ports.(No mutual exclusion on the ports is assigned)

The port capabilities must have Read right.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_PORTID The identifier within the capability is invalid

PORT_TIMEOUT The timeout has elapsed

DELETING_PORT The port has been deleted while waiting for the matching send call.

Errors due to restrictions in the implementation:

EXCEED_MAXCOMW This request exceeds the maximum number of allowed communications waiting for a message in this port

NAME

port_lookup

SYNOPSIS

```
#include <paros.h>

int port_lookup (int service_num, t_portcap *port_cap)
```

DESCRIPTION

The **port_lookup** operation retrieves a capability that was previously registered with the code number **service_num**.

The capability is returned in the structure pointed to by **port_cap**.

Note that the **service_num** is limited by the maximum number of published ports that the system supports at the same time. This limit is set by the constant MAX_PORT_SERV, actually set to 33. So service_num ranges from 0 to 32.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

NOT_BOUND No capability has been registered under service **service_num**.

Errors due to restrictions in the implementation:

ERR_SERV_NUM The requested service number is out of the range of valid service numbers

NAME

port_publish

SYNOPSIS

```
#include <paros.h>

int port_publish (t_portcap port_cap, int service_num)
```

DESCRIPTION

The capability **port_cap** for a given port is registered under code number **service_num**.

The **port_cap** capability must be a capability with Owner right. The capability will be registered without the Owner right.

If a capability for a port is already registered under the service **service_num**, an error is returned.A

Note that the **service_num** is limited by the maximum number of published ports that the system supports at the same time. This limit is set by the constant MAX_PORT_SERV, actually set to 33. So service_num ranges from 0 to 32.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

CNO_RIGHT The capability has not Owner right.

ALREADY_BOUND The service service_num is already associated with a port.

Errors due to restrictions in the implementation:

ERR_SERV_NUM The requested service number is out of the range of valid service numbers

NAME

port_unpublish

SYNOPSIS

```
#include <paros.h>

int port_unpublish (t_portcap port_cap, int service_num)
```

DESCRIPTION

The capability previously bound to service **service_num** is unbound.

Parameter **port_cap** must be a capability with owner right for the same port that was bound to **service_num**, otherwise this call has no effect and

Note that the **service_num** is limited by the maximum number of published ports that the system supports at the same time. This limit is set by the constant MAX_PORT_SERV, actually set to 33. So service_num ranges from 0 to 32.

returns an error.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

CNO_RIGHT Parameter **port_cap** does not have the owner right or it does not correspond to the same port that was registered under **service_num**.

NOT_BOUND No capability has been registered under service **service_num**.

Errors due to restrictions in the implementation:

ERR_SERV_NUM The requested service number is out of the range of valid service numbers

7.6 memory management

This section presents the memory management functions. Some functions in the process and processor management sections (sections 7.3 and 7.4) also include functionalities related to memory management (task creation,...).

NAME

allocate_segment

SYNOPSIS

```
#include <paros.h>

int allocate_segment ( t_Ptaskcap Ptask_cap,
                       t_segment *segment_list,
                       int sl_size)
```

DESCRIPTION

Allocates uninitialized new physical segments (physical address space) to be later mapped in one task logical address space.

Ptask_cap identifies the Ptask to which the segment is allocated. **Segment_list** points to an array of **sl_size** entries. Each entry corresponds to one segment to be created and specifies on which logical processor (**logicalp** field), desired address if one specific base address is required (**base** filed), and length (**len** field).

If the requested **base** address of the segment is 0 the kernel will allocate a segment of the specified length and will return the address. If the **base** address is specified by the user, the kernel will check that the segment does not overlap with already allocated ones.

If a subset of segments can not be created, the field **ident** in the segments that can be created without error will be created. Corresponding entries will contain a negative value on return indicating why they have not been created.

Ptask_cap must have Ptask Manage right.

Note that the **len** field is measured in bytes.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_PTASKID The identifier within the capability is invalid

SOME_ERROR There is at least a segment that was not created, the kind of error is specified at its **ident** field.

NAME

map_segment

SYNOPSIS

```
#include <paros.h>

int map_segment (t_Ptaskcap Ptask_cap, t_taskid task,
                 t_segmentid segment, t_address *plog_addr)
```

DESCRIPTION

Maps the previously created segment of identifier **segment** onto the logical address space of task **task** in the Ptask identified by **Ptask_cap**.

Both segment and task must be on the same logical processor of the Ptask to be correctly mapped. If this is does not hold, an error will be reported.

map_segment returns in the position pointed by **plog_addr** the address within the task logical address space where the segment has been mapped. In the case of transputer based systems this will always correspond to the physical base address of the segment.

Ptask_cap must have Ptask Manage right.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_PTASKID The identifier within the capability is invalid

ERR_TASKID The task identifier is invalid

ERR_SEGMID The segment identifier is invalid

ERR_MAPPED The segment is already mapped

DIFF_LOGPROC The task and the segment are in different logical processors

Errors due to limits in the implementation :

MORE_MAPPED The segment table within the task is full

NAME

unmap_segment

SYNOPSIS

```
#include <paros.h>

int unmap_segment (t_Ptaskcap Ptask_cap, t_segment segment)
```

DESCRIPTION

Unmaps the previously mapped **segment** from the logical address space of a task in the Ptask identified by **Ptask_cap**.

The Ptask capability **Ptask_cap** must have Ptask Manage right.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_PTASKID The identifier within the capability is invalid

ERR_SEGMID The segment identifier is invalid

ERR_UNMAPPED The segment is already unmapped

NAME

copy_segment

SYNOPSIS

```
#include <paros.h>

int copy_segment (t_Ptaskcap src_Ptask, t_segmentid src_seg,
                  long src_offset, long length,
                  t_Ptaskcap dst_Ptask, t_segmentid dst_seg,
                  long dst_offset)
```

DESCRIPTION

Copies **length** bytes at offset **src_offset** from segment **src_seg** of Ptask **src_Ptask** to offset **dst_offset** of segment **dst_seg** of Ptask **dst_Ptask**.

The two Ptask capabilities must have Ptask Segment right.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within some of the capabilities is invalid

CNO_RIGHT Some of the capabilities has not the necessary right

ERR_PTASKID The identifier within some of the capabilities is invalid

ERR_SEGMID At least one segment identifier is invalid

ERR_OFFSET At least one offset is outside the segment

ERR_LENGTH At least one length from the offset exceeds the segment end

ERR_COPY Some error has occurred at PE level

NAME

grow_segment

SYNOPSIS

```
#include <paros.h>

int grow_segment (t_Ptaskcap Ptask_cap,
                  t_segmentid segment_id,
                  int increment, int where)
```

DESCRIPTION

Increases the length of segment **segment_id** in Ptask **Ptask_cap** by the amount **increment**(can be positive for increase or negative for decrease). The increase will be done at the BEGINNING or at the END of the segment depending on the value of where.

If the physical memory that is requested is occupied by other segment an error is returned.

The Ptask capability must have Ptask Manage right.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_PTASKID The identifier within the capability is invalid

ERR_SEGMID The segment identifier is invalid

NO_FREE_MEM The segment can not be enlarged because the physical memory behind it has already been allocated.

STATUS

This call is not supported in Paros Free Distribution v1.2.

NAME

which_segment

SYNOPSIS

```
#include <paros.h>

int which_segment (t_address addr,
                   t_segmentid *segment_id,
                   long *offset)
```

DESCRIPTION

Given an address **addr** returns the identifier of the segment it belongs to and the offset from the base address.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

ERR_ADDR The address is invalid

NAME

delete_segment

SYNOPSIS

```
#include <paros.h>

int delete_segment (t_Ptaskcap Ptask_cap,
                    t_segmentid segment_id)
```

DESCRIPTION

The segment **segment_id** within Ptask **Ptask_cap** is deleted.

If the segment is mapped to a task it is unmapped before deleting it. If there is any thread of the task running with its stack in this segment, the call returns an error and the segment is unaffected.

Ptask_cap must have Ptask Manage right.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_PTASKID The identifier within the capability is invalid

ERR_SEGMID The segment identifier is invalid

SEGMENT_BUSY There is a running task with its stack mapped on to this segment.

SEE ALSO

segment_create, map_segment, unmap_segment

7.7 protection management

This section presents the interface functions to modify capabilities. A full capability for an object is obtained by its creator. Revocations functions based on total invalidation are also available.

NAME

cap_refine

SYNOPSIS

```
#include <paros.h>

int cap_refine (t_objecttype object_type,
                t_cap old_cap,
                t_rightmask mask,
                t_cap *new_cap)
```

DESCRIPTION

This call returns a new capability for the object of type **object_type** with less rights than the original. The **mask** parameter specifies the rights in the old capability **old_cap** that will not be present in the new one to be returned in the structure pointed by **new_cap**.

The possible rights and types are defined in file *paros.h*.

There is no way to increase the rights of a capability.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_CLUSTERID The identifier within the cluster capability is invalid

ERR_PTASKID The identifier within the Ptask capability is invalid

ERR_PORTID The identifier within the Port capability is invalid

NAME

new_key

SYNOPSIS

```
#include <paros.h>

int new_key (t_objecttype object_type,
             t_cap old_cap,
             t_cap *new_cap)
```

DESCRIPTION

The kernel returns a new capability for the object of type **object_type**, invalidating all previous capabilities. The old capability **old_cap** must have Owner right. The new capability is returned in the structure pointed by **new_cap** and has all rights on the object.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

KEY_INVAL The key within the capability is invalid

CNO_RIGHT The capability has not the necessary right

ERR_CLUSTERID The identifier within the cluster capability is invalid

ERR_PTASKID The identifier within the Ptask capability is invalid

ERR_PORTID The identifier within the Port capability is invalid

Chapter 8

Library functions

This chapter describes the library functions that are provided with the Paros Free Distribution Kernel v1.2 as support both for parallelism and classical heavy weight OS functionalities.

These functions have been totally implemented on top of the Paros Kernel interface and provide more powerful functionalities to the application developer. In this way the Kernel is left very general and the library functions offer a more particular environment.

We differentiate five groups of functions, the first one describes the services offered through subsystem servers to the application programmer, the next one includes miscellaneous functions that are given as examples of the great flexibility of the Paros Kernel. The third group describe the services offered by the graphic server. These group of functions are only valid for PFD v1.2 with DOS as host operating system. In the fourth and fifth group, the support for the Inmos C and 3L Fortran programming models are described. Both examples demonstrate that existing programming models and environments can be implemented and coexist on top of an Operating System Micro-kernel.

The sources of some of these functions are also given (although some parts of it are dependent on subsystem conventions such as those used by the loader) in the Free Distribution v1.2. They are examples of how functions of very high semantic level can be implemented on top of the Paros Kernel interface.

All the functions, excluding those related with Fortran, have been grouped in a single library named *user.lib*. Fortran interface has been grouped in another library called *fortran.lib*. If you want to build an application using Fortran language, you need to link with both libraries, otherwise you only need to link with *user.lib*.

8.1 Subsystem Access libraries

In this section, we list the libraries offered to access typical heavy weight Operating System functions that are supported in Paros Free Distribution v1.2 by subsystem level servers.

The UNIX kernel functions *open, close, read, write, lseek* are offered by a Paros file server that provides access to files on the host.

The standard I/O library is offered to applications tasks. The *print* functions offer exclusion between threads of the same tasks, so their output don't interfere each other. Threads of different task don't need exclusion because they don't share I/O buffers.

The *malloc* function is also available in PFD v1.2. It operates out of a Paros segment that, if needed, is allocated and mapped to the task by the library at run time. A default size for this heap segment is 64Kbytes. In case an application foresees that it will need a bigger heap, the function *malloc_init* must be invoked prior to any memory allocation to set the size of this segment.

The functions *stdio_dup* and *stdio_cancel* have been added to the PFD v1.2 in order to allow a stronger interaction between the file server and other subsystems, such as shells. These functions are used to duplicate or update all the information related with file descriptors and working directories.

Often, it may be necessary to read data from files and then copy that data from one buffer to another (probably in different segments). This results with low performance caused by data transfers. In order to avoid unnecessary data movements, a new function *remote_read* ,has been implemented. This function allows to read from a file and specify the destination segment.

The Unix *stat* function is described in this section, as it does not offer the same interface and result as the original call. The rest of the standard Unix functions will not be described, refer to 3LC manual.

NAME

subsys.h

SYNOPSIS

```
extern int malloc_init   ( int );
extern int remote_read   ( int, t_Ptaskcap, int, int, int );
extern int stdio_dup     ( t_Ptaskcap,t_Ptaskcap,int,int,int,char *,
                           char *);
extern int stdio_cancel  ( t_Ptaskcap );
```

NAME

sys/stat.h

SYNOPSIS

```
#define ST_DIR  0       /* is directory */
#define ST_FILE 1

typedef struct  /* struct for stat call */
{
  long mod_time;        /* time of last modification */
  int  type;            /*  indicates type of the file */
}t_file_stat;

extern int stat(char * ,t_file_stat *);
```

NAME

malloc_init

SYNOPSIS

```
#include <paros.h>
#include <subsys.h>

int malloc_init (int n_bytes)
```

DESCRIPTION

Initialize the heap to the **n_bytes** specified, if it is done before any memory space allocation, otherwise (if *malloc* was called first, setting the heap to the default value 64K), **malloc_init** returns an error.

It is highly recommended to terminate an application (with *exit_terminate*) after *malloc_init* errors.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

If no sufficient space is available it returns -1.

If memory allocation was performed previously, this function returns also -1.

NAME

remote_read

SYNOPSIS

```
#include <paros.h>
#include <subsys.h>

int remote_read( int fdes, t_Ptaskcap Ptask_dest,
                 int seg_dest, int offset, int size);
```

DESCRIPTION

This function reads **size** bytes from a file, whose file descriptor is **fdes**. The data read, will be directly copied in the segment **seg_dest** at position **offset** of the Ptask **Ptask_dest**. No temporary buffer in the calling process is needed. The file server reads and copies directly to the destination segment.

RETURN VALUE

Upon successful completion, the operation returns the size read in bytes.

When an error is detected (file not open, error reading, error copying) it returns -1.

NAME

stdio_dup

SYNOPSIS

```
#include <paros.h>
#include <subsys.h>

int stdio_dup( t_Ptaskcap father_Ptask, t_Ptaskcap user_Ptask,
               int std_in, int std_out, int std_err, char *cwd,
               char *hd);
```

DESCRIPTION

This function is used to update descriptors and working directories of the Ptask **user_Ptask**. In case **std_in, std_out** or **std_err** are equal to -1, the Ptask **user_Ptask** will inherit the correspondingly standard descriptors of the Ptask **father_Ptask**. Any other value is consider as a new opened descriptor, which will be used as standard input, output or error, respectively, for the **user_Ptask**. A null string ("") in **cwd** and/or **hd**, indicates that **user_Ptask** will inherit the current working directory and/or the home directory from **father_Ptask**. Any other string will be considered as the new current working directory and/or home directory of **user_Ptask**.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

If an internal error takes place -1 is returned.

NAME

stdio_cancel

SYNOPSIS

```
#include <paros.h>
#include <subsys.h>

int stdio_cancel ( t_Ptaskcap Ptaskcap);
```

DESCRIPTION

This function indicates to the file server that standard input, output and error files of Ptask **Ptaskcap** must be closed. Files will only be closed in case no other Ptask is using these files.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

If an internal error takes place -1 is returned.

NAME

stat

SYNOPSIS

```
#include <paros.h>
#include <sys/stat.h>

int stat (char * filename, t_file_stat * file_stat)
```

DESCRIPTION

This function returns information about the file **filename** in the structure pointed by **file_stat**. Actually, the information returned includes the time of the last modification of the file (field **mod_time** of the t_file_stat structure) and the type of the file (field **type**).

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure it returns one of the following values:

ENOTDIR A component of the path prefix is not a directory.

ENOENT The named file does not exist.

EACCES Search permission is denied for a component of the path prefix.

EFAULT filename points to an invalid address.

8.2 Miscellaneous parallel library

In this section we describe some functions implemented on top of the Paros interface that provide more powerful functionalities to the application developer. These include functions for creation of local threads, some functions from the stand alone 3L C environment, functions used to load executable files directly from an application and functions that support user controlled process migration. The sources of some of these functions are also given in the Free Distribution v1.2 as examples of how functions of very high semantic level can be implemented on top of the Paros Kernel interface, although some parts of it are dependent on subsystem conventions such as those used by the loader.

The functions from the stand alone 3L C environment that are also available are those defined in header files:

string.h
stdlib.h
math.h
sema.h

The sources for some of the functions (*local_fork*, *where_am_i*) are given in directory /Paros/user/source.

NAME

Misc_lib.h

SYNOPSIS

```
#define INFO_OK          0
#define INFO_NOT_VALID   1

extern int  lib_loc_th_create (int (*)(), t_wptr, int, int, ...);
extern int  lib_loc_th_detach (int (*)(), t_wptr, int, int, ...);
extern void lib_wait_loc_th   (void);
extern int  lib_load_file (char *, int, char **, t_Ptaskcap *);
extern int  lib_load_file_in_Ptask (char *, int, char **, t_Ptaskcap,
                                    t_logprocessorid);
extern int  lib_load_file_in_task  (char *, int, char **, t_Ptaskcap,
                                    t_taskid taskid, t_logprocessorid);
extern void where_am_I        (t_processor *, t_processor *);
extern int  local_fork        (void);
extern int  get_task_index    (int *);
extern int  get_num_tasks     (int *);
extern int  get_log_proc      (t_logprocessorid *);
extern void clock_delay       (int);
extern int  get_time          (void);
```

NAME

lib_loc_th_create

SYNOPSIS

```
#include <paros.h>
#include <misc_lib.h>

int lib_loc_th_create (void *function, void *ws,
                       int wssize, int nb_params,...)
```

DESCRIPTION

lib_loc_th_create is a library function that directly (without kernel intervention) creates a new ligthweigth thread. The first parameter specifies the function that will be executed by the thread. The second and third specify the workspace (stack) that will be used by the thread and its size (in integers) respectively. Next is the number of parameters passed to the function to be started, followed by the values that will be passed as parameters.

lib_loc_th_create keeps a single global counter of threads created by this call, which is also used by *lib_wait_loc_th*. Several levels of structured co-begin/co-end are thus not supported by these calls. A thread created by this call terminates when the routine returns. It can also invoke the *exit* or *exit_terminate* functions, which kill the task or the whole application respectively. Using the kernel *stopp* function, it kills the thread, but neither the counter of threads nor the task or Ptask status are affected.

RETURN VALUE

This function always returns OK.

NAME

lib_wait_loc_th

SYNOPSIS

```
#include <paros.h>
#include <misc_lib.h>

void lib_wait_loc_th()
```

DESCRIPTION

The *lib_wait_loc_th* function waits until all the threads created through the *lib_loc_th_create* call finish. There is an implicit call to *lib_wait_loc_th* after the main function of a task returns.

RETURN VALUE

This function always returns OK.

NAME

lib_loc_th_detach

SYNOPSIS

```
#include <paros.h>
#include <misc_lib.h>

int lib_loc_th_detach (void *function, void *ws,
                       int wssize, int nb_params,...)
```

DESCRIPTION

This library function performs exactly the same as the function *lib_loc_th_create* but the thread, in this case, is detached from its creator. It is not possible to wait for the ending of the thread execution.

RETURN VALUE

This function always returns OK.

NAME

local_fork

SYNOPSIS

```
#include <paros.h>
#include <misc_lib.h>

int local_fork()
```

DESCRIPTION

local_fork is similar to the UNIX fork within the context of a Ptask. It creates a new task within the same Ptask on another logical processor. The father and son tasks have the same memory image. *local_fork* returns a 0 value to the son and 1 to the father tasks.

RETURN VALUE

local_fork returns a 0 value to the son and 1 to the father tasks. In case some error occurs in the kernel calls invoked by *local_fork*, it will return the error received. The functions called are :

get_Ptask_cap
get_task_id
task_info
allocate_processors
task_create
copy_segment
port_create
Thread_create

NAME

where_am_I

SYNOPSIS

```
#include <paros.h>
#include <misc_lib.h>

void where_am_I(t_processor *ph_proc, t_processor *log_proc)
```

DESCRIPTION

This function returns in the position pointed by **log_proc** the logical processor number where the caller thread is running, and in the position pointed by **ph_proc** the physical processor number where this logical processor has been mapped.

NAME

get_timer

SYNOPSIS

```
#include <paros.h>
#include <misc_lib.h>

int get_timer(void)
```

DESCRIPTION

Returns the local clock (rated at one increment every 64 microseconds).

NAME

clock_delay

SYNOPSIS

```
#include <paros.h>
#include <misc_lib.h>

void clock_delay(int time)
```

DESCRIPTION

Blocks the current thread for **time***64 microseconds ticks.

NAME

get_task_index

SYNOPSIS

```
#include <paros.h>
#include <misc_lib.h>

int get_task_index(int *pindex);
```

DESCRIPTION

Returns, in the position pointed by **pindex**, the number of the task according to the configuration file, when working with CSP programming model. In case SPMD programming model is used, each task of the Ptask will be given an index which ranges from 0 to the total number of tasks minus one. **pindex** will contain this index. In case the task was started dynamically by the library calls *lib_load_file*, *lib_load_file_in_Ptask* and *lib_load_file_in_task*, the index of the task is set to the total number of existing tasks minus one. If the task was dynamically created at run time, without using those library calls, this information will not be valid. In those cases the function returns **INFO_NOT_VALID**.

Recall that this information is set when starting the task,without notifying the kernel and is not updated during run time.

RETURN VALUE

Upon successful completion, the operation returns **INFO_OK**

In case of failure, it returns **INFO_NOT_VALID**

NAME

get_num_tasks

SYNOPSIS

```
#include <paros.h>
#include <misc_lib.h>

int get_num_tasks(int *nb_tasks);
```

DESCRIPTION

Returns, in the position pointed by **nb_tasks**, the total number of tasks that where defined in the configuration file when working with CSP programming model. If the task was started within a SPMD program environment, this function will return the total number of tasks that were started. If the task was dynamically created by the library calls *lib_load_file*, *lib_load_file_in_Ptask* and *lib_load_file_in_task*, **nb_tasks** will contain the actual number of existing tasks. If the task was dynamically created by any other call (*task_create*) , this information will not be valid. In those cases the function returns **INFO_NOT_VALID**.

Recall that this information is set when starting the task,without notifying the kernel and is not updated during run time.

RETURN VALUE

Upon successful completion, the operation returns **INFO_OK**

In case of failure, it returns **INFO_NOT_VALID**

NAME

get_log_proc

SYNOPSIS

```
#include <paros.h>
#include <misc_lib.h>

int get_log_proc(t_logprocessorid *log_proc);
```

DESCRIPTION

Returns, in the position pointed by **log_proc**, the logical processor where the calling process was mapped to. This information will not be valid in case the task was dynamically created at run time, without using the following library calls:*lib_load_file*, *lib_load_file_in_Ptask* and *lib_load_file_in_task*.

RETURN VALUE

Upon successful completion, the operation returns **INFO_OK**

In case of failure, it returns **INFO_NOT_VALID**

NAME

lib_load_file

SYNOPSIS

```
#include <paros.h>
#include <misc_lib.h>

int lib_load_file(char *filename,
                  int argc,
                  char **argv,
                  t_Ptaskcap *Ptask_cap)
```

DESCRIPTION

lib_load_file is a library function that will create a Ptask, with one task, which will execute the code in **filename**. Only one thread will be started. This function will create and allocate the necessary resources (Ptask, task, segments, logical processor, thread). The task will be created in the first free logical processor of the new Ptask , which will be logical processor 0. This Ptask will run independently from its creator, that means that the new Ptask will continue its execution even though its creator finishes executing and dies. The function will return the Ptask capability (**Ptask_cap**) of the created Ptask. Arguments can be passed to the thread that will execute the code in **filename** in the standard Unix way. **argc** indicates the number of parameters that will be found in the arrays of pointers to arguments **argv**.

The new created Ptask inherits the standard input/output from its creator. Input/output can not be redirected at this level.

RETURN VALUE

Upon successful completion, the operation returns the identifier of the created task.

In case some error occurs in the kernel calls invoked by *lib_load_file*, it will return the error received. The functions called are :

Ptask_create
allocate_segment
task_create
copy_segment
I/O functions

Thread_create
Ptask_info

NAME

lib_load_file_in_Ptask

SYNOPSIS

```
#include <paros.h>
#include <misc_lib.h>

int lib_load_file_in_Ptask(char * filename, int argc,
                           char ** argv, t_Ptaskcap Ptask_cap,
                           t_logprocessorid log_proc)
```

DESCRIPTION

lib_load_file_Ptask is a library function that will create a task, local to the Ptask **Ptask_cap**, which will execute the code in **filename**. The task will be created on the logical processor **log_proc**. Only one thread will be started. This function will create and allocate the necessary resources (task, segments, thread). Arguments can be passed to the thread that will execute the code in **filename** in the standard Unix way. **argc** indicates the number of parameters that will be found in the arrays of pointers to arguments **argv**.

Input/output can not be redirected at this level. The call *stdio_dup* can be used to redirect input/output.

Note that the Ptask specified (**Ptask_cap**) can also be the Ptask of the calling process.

RETURN VALUE

Upon successful completion, the operation returns the identifier of the created task.

In case some error occurs in the kernel calls invoked by *lib_load_file_in_Ptask*, it will return the error received. The functions called are :

allocate_segment
task_create
copy_segment
I/O functions
Thread_create
Ptask_info

NAME

lib_load_file_in_task

SYNOPSIS

```
#include <paros.h>
#include <misc_lib.h>

int lib_load_file_in_task(char * filename,
                          int argc,
                          char ** argv,
                          t_Ptaskcap Ptask_cap,
                          t_taskid taskid,
                          t_logprocessorid log_proc)
```

DESCRIPTION

lib_load_file_task is a library function that will create a thread in the task specified by **taskid**, local to the Ptask **Ptask_cap**, which will execute the code in **filename**. Only one thread will be started. The task **taskid** must be previously created on the logical processor **log_proc**. No other resources needs to be allocated or created. This function will create and allocate the necessary resources (segments, thread). Arguments can be passed to the thread that will execute the code in **filename** in the standard Unix way. **argc** indicates the number of parameters that will be found in the arrays of pointers to arguments **argv**.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case some error occurs in the kernel calls invoked by *lib_load_file_in_task*, it will return the error received. The functions called are :

allocate_segment
map_segment
copy_segment
I/O functions
Thread_create
Ptask_info

8.3 Graphic library functions

In this section, we describe the graphics functions that are supported in Paros Free Distribution v1.2 by the graphic server.

In PC based hardware, a VGA/CGA display server has been implemented, which offers the possibility to draw dots, lines and rectangles on the screen in both cga and vga mode. Graphics can also be combined with text in vga mode.

The library functions, that can be called from the user code, are complemented by some commands that the command line interpreter executes directly (*clear*, *mode*).

The display server offers the screen as a shared device to which several applications can draw concurrently. It is implemented as a multi-threaded server. All threads receive from a port that is published.

NAME

vga.h

SYNOPSIS

```
#define VGA_MODE        0x12
#ifndef TEXT_MODE
#define TEXT_MODE       0x2
#endif

#define VGA_XMAX        639
#define VGA_YMAX        479

#define BLACK           0
#define BLUE            1
#define GREEN           2
#define CYAN            3
#define RED             4
#define MAGENTA         5
#define BROWN           6
#define LIGHTGRAY       7
#define DARKGRAY        8
#define LIGHTBLUE       9
#define LIGHTGREEN     10
#define LIGHTCYAN      11
#define LIGHTRED       12
#define LIGHTMAGENTA   13
#define YELLOW         14
#define WHITE          15

/*graphics functions*/
extern  int     change_mode (int);
extern  int     clear_screen();
extern int     vga_mode (int);
extern int     vga_plot (int, int, int);
extern int     vga_line (int, int, int, int, int);
extern int     vga_bar (int, int, int, int, int, int, int);
extern int     vga_text (char, int, int, int, int);
extern int     vga_erase_bar (int, int, int, int);


/* the parameters of the graphics functions are:
int     vga_mode (int);
```

```
int     vga_plot (x, y, color);
int     vga_line (x_inf, y_inf, x_sup, y_sup, color);
int     vga_bar  (x_inf, y_inf, x_sup, y_sup, border_col, ,fill,fill_col);
int     vga_text (char, x, y, fg, bg);
int     vga_erase_bar (int x_inf, int y_inf, int x_sup, int y_sup);
*/
```

NAME

cga.h

SYNOPSIS

```
#ifndef TEXT_MODE
#define TEXT_MODE                 0x02
#endif

#define MONO_80COL_TEXT_MODE      0x02
#define CGA_LORES_GRAPHICS_MODE   0x05
#define CGA_HIRES_GRAPHICS_MODE   0x06
#define DONT_CLEAR_DISPLAY        0x80

#define CGAL_XMAX 319
#define CGAL_YMAX 199

#define CGAH_XMAX 639
#define CGAH_YMAX 199

/*graphics functions*/
extern int cga_mode (int);
extern int cga_lores_plot (int, int , int);
extern int cga_lores_line (int, int , int, int, int);
extern int cga_lores_bar (int, int , int, int, int, int);
extern int cga_lores_erase_bar (int, int , int, int, int);
extern int cga_hires_plot (int, int , int);
extern int cga_hires_line (int, int , int, int, int);
extern int cga_hires_bar (int, int , int, int, int, int);
extern int cga_hires_erase_bar (int, int , int, int, int);

/* the parameters of the graphics functions are:
extern int cga_mode (mode 5 o 6);
extern int cga_lores_plot (x, y , color);
extern int cga_lores_line (x_inf, y_inf , x_sup, y_sup, color);
extern int cga_lores_bar (x_inf, y_inf, x_sup, y_sup, color, fill);
extern int cga_lores_erase_bar (x_inf, y_inf, x_sup, y_sup, fill);
extern int cga_hires_plot (x, y, color);
extern int cga_hires_line (x_inf, y_inf, x_sup, y_sup, color);
extern int cga_hires_bar (x_inf, y_inf, x_sup, y_sup, color, fill);
extern int cga_hires_erase_bar (x_inf, y_inf, x_sup, y_sup, fill);
*/
```

NAME

change_mode

SYNOPSIS

```
#include <vga.h>

int change_mode(int mode);
```

DESCRIPTION

Changes the current screen mode to the **mode** specified. Paros supports the following screen modes: 0x2, 0x5, 0x6 and 0x12.

RETURN VALUE

Upon successful completion, the operation returns **OK**.

In case of failure, it returns the following value:

INVALID_MODE If the mode specified is not supported by the system.

NAME

clear_screen

SYNOPSIS

```
#include <vga.h>

int clear_screen()
```

DESCRIPTION

Clears the screen without changing the current mode.

RETURN VALUE

In case the kernel functions it calls (port_lookup, port_create, port_send and port_receive) fail it will return the obtained error, otherwise it returns **OK**.

NAME

vga_mode

SYNOPSIS

```
#include <vga.h>

int vga_mode(int mode);
```

DESCRIPTION

Changes the current mode of the video screen to the **mode** specified, clearing the screen if the last mode was different from the new one. To get the standard and correct vga mode (640x480 with 16 colors) use mode 18 (or 0x12).

RETURN VALUE

In case the kernel functions it calls (port_lookup, port_create, port_send and port_receive) fail it will return the obtained error, otherwise it returns **OK**. If the *mode* specified is incorrect, it returns **INVALID_VGA_MODE**.

NAME

vga_plot

SYNOPSIS

```
#include <vga.h>
int vga_plot (int x, int y, int color):
```

DESCRIPTION

Draws a point with the specified **color**.

RETURN VALUE

This function may call port_lookup, port_create, port_send and port_receive. In case of failure it will return the obtained error, otherwise it returns **OK**.

NAME

vga_line

SYNOPSIS

```
#include <vga.h>
int     vga_line (int x_inf, int y_inf, int x_sup, int y_sup, int color):
```

DESCRIPTION

Draws a line with the specified **color**.

RETURN VALUE

This function may call port_lookup, port_create, port_send and port_receive.
In case of failure it will return the obtained error, otherwise it returns **OK**.

NAME

vga_bar

SYNOPSIS

```
#include <vga.h>
int     vga_bar (int x_inf, int y_inf, int x_sup, int y_sup,
             int border_color, int filled, int fill_color):
```

DESCRIPTION

Draws a bar with the **border_color** specified. The **filled** flag indicates whether it has to be filled (1) or empty(0). In case it is filled, it will be drawn with the **fill_color**.

RETURN VALUE

This function may call port_lookup, port_create, port_send and port_receive. In case of failure it will return the obtained error, otherwise it returns **OK**.

NAME

vga_text

SYNOPSIS

```
#include <vga.h>
int     vga_text (char, int x, int y, int fg_color,
                  int bg_color):
```

DESCRIPTION

Draws a character with the specified background and foreground colors (**bg_color** and **fg_color** respectively). The coordinates *x* and *y* refers to the upper-left corner. The size occupied by the characters are of 8x8 dots.

RETURN VALUE

This function may call port_lookup, port_create, port_send and port_receive. In case of failure it will return the obtained error, otherwise it returns **OK**.

NAME

vga_erase_bar

SYNOPSIS

```
#include <vga.h>

int vga_erase_bar (int x_inf, int y_inf,
                   int x_sup, int y_sup);
```

DESCRIPTION

Erases a bar.

RETURN VALUE

This function may call port_lookup, port_create, port_send and port_receive. In case of failure it will return the obtained error, otherwise it returns **OK**.

NAME

cga_mode

SYNOPSIS

```
#include <cga.h>

int cga_mode(int);
```

DESCRIPTION

Changes the current mode of the video screen to the **mode** specified. To get the low resolution mode (320x200 with 4 colours) use mode 5. To get high cga resolution (640x200 with 2 colors) use mode 6.

RETURN VALUE

This function may call port_lookup, port_create, port_send and port_receive. In case of failure it will return the obtained error, otherwise it returns **OK**. In case the *mode* specified is incorrect, it returns **INVALID_CGA_MODE**.

NAME

cga_plot

SYNOPSIS

```
#include <cga.h>

int     cga_lores_plot (int x, int y, int color);
int     cga_hires_plot (int x, int y, int color);
```

DESCRIPTION

Draws a dot with the specified **color**, in high/low resolution.

RETURN VALUE

These functions may call port_lookup, port_create, port_send and port_receive. In case of failure it will return the obtained error, otherwise it returns **OK**.

NAME

cga_line

SYNOPSIS

```
#include <cga.h>
int      cga_lores_line (int x_inf, int y_inf, int x_sup,
                         int y_sup, int color):
int      cga_hires_line (int x_inf, int y_inf, int x_sup,
                         int y_sup, int color):
```

DESCRIPTION

Draws a line with the specified **color**.

RETURN VALUE

These functions may call port_lookup, port_create, port_send and port_receive. In case of failure it will return the obtained error, otherwise it returns **OK**.

NAME

cga_bar

SYNOPSIS

```
#include <cga.h>
int cga_lores_bar (int x_inf, int y_inf, int x_sup, int y_sup,
                   int border_color, int filled, int fill_color):
int cga_hires_bar (int x_inf, int y_inf, int x_sup, int y_sup,
                   int border_color, int filled, int fill_color):
```

DESCRIPTION

Draws a bar with the **border_color** specified. The **filled** flag indicates whether it has to be filled (1) or empty(0). In case it is filled, it will be drawn with th **fill_color**.

RETURN VALUE

These functions may call port_lookup, port_create, port_send and port_receive. In case of failure it will return the obtained error, otherwise it returns **OK**.

NAME

cga_erase_bar

SYNOPSIS

```
#include <cga.h>

int     cga_lores_erase_bar (int x_inf, int y_inf, int x_sup,
                             int y_sup);
int     cga_hires_erase_bar (int x_inf, int y_inf, int x_sup,
                             int y_sup);
```

DESCRIPTION

Erases a bar in either high or low resolution.

RETURN VALUE

These functions may call port_lookup, port_create, port_send and port_receive. In case of failure it will return the obtained error, otherwise it returns **OK**.

8.4 Inmos Compatibility Library

This section lists the subset of the Inmos C run time libraries that have been implemented on top of Paros. This is accompanied by the utility *cubata* (described in subsection 9.2.5). All together it leads to very fast portability of Inmos C applications developed for raw transputers to Paros.

The Ilib library thus supports the Inmos C programming model on top of Paros, with the additional advantage that all the Paros features (mapping to physical resources at run time, access to file and display servers from any processor, possibility to load several applications concurrently, ...) are also available to the programmers already experienced in this model of parallel programming.

The list of Inmos C functions that have been implemented on top of Paros are: *ChanAlloc, ChanIn, ChanOut, ProcAlloc, ProcAltList, ProcParList, ProcRun, exit* and *exit_terminate*.These last two functions are described in this section, refer to the Inmos manual for a description of the rest.

We must point out that the call **ProcAlloc** allocates memory for the processes stack. The size allocated is equal to the stack size (considering it to be a stack of integers) plus the size of the Inmos structure *Process*. This must be taken into account when the heap is initialized.

NAME

exit

SYNOPSIS

```
void exit(int status)
```

DESCRIPTION

The task of the thread that make this call will be killed. All the threads that belongs to that task will be killed and the resources used will be freed. The process calling will never return from this call.

NAME

exit_terminate

SYNOPSIS

```
void exit_terminate (int status)
```

DESCRIPTION

The Ptask of the thread that make this call will be killed. All the tasks of that Ptask will be killed and the resources used will be freed.

8.5 Fortran Library

The possibility to run Fortran programs that use the 3L Fortran library are included in Paros Free Distribution v1.2. This allows fast portability of 3L Fortran applications to Paros with the additional advantage of Paros features availability. For that purpose a small interface is offered and some examples have been included. The Fortran 3L functions available on top of PFD v1.2 are the following:

```
f77_chan_out_byte
f77_chan_in_byte
f77_chan_out_word
f77_chan_in_word
f77_chan_out_message
f77_chan_in_message
f77_chan_in_port
f77_chan_out_port

f77_thread_start
f77_thread_create
f77_thread_stop
f77_thread_deschedule

f77_alt_wait
f77_alt_wait_vec

f77_timer_now

f77_sema_init
f77_sema_wait
f77_sema_signal
f77_sema_wait_n
f77_sema_signal_n
f77_sema_test_wait

f77_r_is_nan
f77_d_is_nan
f77_r_is_finite
f77_d_is_finite
```

Mathematical functions are also offered. Reefer to 3L Fortran manual for a description of these functions.

Fortran input/output is not supported. For that reason the user has to implement any I/O function in C language and call them from the Fortran program. The advantage is that all the tasks can perform I/O operations, no matter their location. We provide an example that can be used as reference (see in directory /paros/user/fortran for file pinta.c). We also provide the interface for file handling (open, close, read and write) (see file file.c also in directory /paros/user/fortran).

The semantics of functions *f77_chan_in_port* and *f77_chan_out_port* have been slightly modified with respect to the 3L Fortran definition. On top of Paros no distinction is made between input and output channel lists. Thus both functions are identical and the parameter corresponds to the index of the channel in the single list specified in the Paros configuration file. The user should know for each index, whether it corresponds to an input or output channel. When using any of the 3L Fortran functions remind to include the corresponding function definition files (*.inc).

Regarding Paros functions, no interface has been made, so the user must be aware that fortran functions parameters are always passed by reference, and many Paros functions have some parameters passed by values, so they can not be called directly, and an interface must be done. I/O functions can be useful as a model to follow to perform any interface.

See section 6.5 for Fortran examples.

Part V

Paros Reference Users Guide

Chapter 9

Utilities

This chapter describes some utilities and tools that are provided with the Paros Free Distribution v1.2 that may be useful for application development and tuning. Some of this utilities are prepared to be run on top of Paros and some without Paros.

For the moment, they are grouped in three basic types: user commands, configuration and application building and monitoring tools.

9.1 User Commands

When Paros boots, a Command Line Interpreter (Cli) starts its execution on top of Paros Kernel, scanning the input stream for commands. This command line interpreter runs on to the root processor as well as the file server. It offers simple commands, but it allows to construct more complex ones on top of it. Commands may be **built-in** commands or **external** commands.

9.1.1 Cli Built-in commands

Built-in commands are executed within the command line interpreter and they are:

cd [*path*]; changes current directory (no default home directory). When Paros boots the initial directory is, by default, */paros/user*.

clear returns screen to normal text mode in case an application leaves it in a graphical mode.

echo [*strings*]

edit [*filename*]; calls your selected editor. See appendix A for installation.

exit

help lists all the available commands.

kill *pid*; kill the Ptask with the process identifier *pid*.

ls trivial command for file listing.

mode changes screen mode. Only offered for version with DOS as Host OS.

more *filename*; trivial more command. It can not display a file that is still opened by an application (host server restriction).

path [*new_path*]; shows and changes the alternative path where the command line interpreter also searches if the command specified is not a Built-in command and it was not found in the current directory. In this version, only one search path can be specified at a time.

ps [*number*] [**-m**]; displays information about the number of Paros applications loaded by the command line interpreter. If a given number of application is requested, it display information about its logical structure and mapping to physical processors. If the **-m** option is given it displays the base addresses of the segments that constitute the application.

pwd shows current directory.

quit

shell temporary exit to the booting operating system.

sleep *seconds*; blocks the *cli* the time specified in *seconds*.

wait [*pid*]; blocks the *cli* waiting for the specified Ptask to end. If no *pid* number is specified the command line interpreter will wait for all the actual running Ptasks to end.

? same as help.

Built-in commands can not be redirected.

Note, that in Paros Free Distribution v1.2, commands *linkalos* and *cubata* have been moved from Built-in commands to Non Built-in commands.

9.1.2 Cli Non Built-in commands

Non Built-In commands may be of three types: **executable_files, configuration_files** or **scripts**.

If the command corresponds to an **executable_file**, the command line interpreter loads it as a sequential application (a Ptask with only one task and one thread executing in it).

If the command corresponds to a **configuration_file**, a Ptask with the logical structure specified in the configuration file is loaded.

In both cases, *parameters* can be passed from the command line interpreter to the application and the standard input and output can be redirected in the classical UNIX syntax. Parameters from the command line are passed to all the tasks of a parallel application. The standard input/output are the same for all the tasks of an application.

If the command corresponds to a **script_file**, the command line interpreter closes its standard input and starts to read commands from the new specified file. When there are no more commands in the file, the user terminal is opened again. The script files need to have as first line the string : **#paros**. Script files can interpret any of the built-in and non built-in commands described previously. You can even call another script file from it, but only one level of overlapping is permitted (same as in DOS). Parameters may also be passed to a script file. See section 6.4 for more details.

The format of a configuration file in a BNF like syntax is

conf ::= Ptask description

number of processors : *number allocation*
number of tasks : *number*
number of channels : *number*
number of executable files : *number*
{*exec_mappings*}
[*list_of_channels*]
tasks_parameters

allocation ::= logical | physical

exec_mappings ::= use *exec_file* in *number* task
{ T*number* in logical processor *number* }

list_of_channels ::= declare channels
{ C*number* from T*number* to T*number* }

task_parameters ::= Task parameters
{ T*number* ({C*number* }) }

If the specified allocation is **logical**, the actual mapping of logical to physical processors will depend on the availability of resources at execution time. The kernel may group several logical processors on to a single physical processor.

If the specified allocation is **physical** the exact number of requested processors will be allocated for exclusive use of this Ptask. Each logical processor will be mapped to a different physical processor. In the case not enough free physical processors are available, the application will not be loaded and an error will be reported.

9.2 Configuration and application building

This section shows, with details, the necessary tools to edit a file, the tools to generate executable files and Paros configuration files. For each step, the commands implemented on top of Paros, and those implemented on top of the Host operating system are shown.

9.2.1 Editing

NAME

edit

SYNOPSIS

edit *file_name*

DESCRIPTION

To generate the source code of your application, it is possible to edit from Paros environment, without having to exit PFD system v1.2. To install your favorite editor see appendix A for a complete explanation.

9.2.2 Compiling

The compilation phase can be performed both on top of Paros and on top of the Host Operating System. The file obtained will have the extension **.bin**.

Compiling on top of Paros

To compile on top of Paros, we provide with this version script files both to compile C and Fortran programs. As you can see in these script files, the compilers are started from an afserver prepared to run on top of Paros.

NAME

comp

SYNOPSIS

comp *file_name*

DESCRIPTION

comp is a script file prepared to compile programs written in C. **file_name** must be the name of a C code file. The filename must be specified without extension.

comp will be found in directory */paros/bin* with the following contents:

```
#paros
\paros\bin\afserver.sn -:o 1 -:b \tc2v1\tc.b4 $1.c
                    /I\paros\include /T8 /X
```

DIAGNOSTIC

In order to improve the compilation step, some temporary files are created on top of the transputers memory. That limits the size of the files, so for some big files it may not be possible to compile on top of Paros. In the next version of Paros, a cache for the file server will be implemented so temporary files will not be necessary, allowing bigger and faster compilations on top of Paros.

NAME

compf

SYNOPSIS

compf *file_name*

DESCRIPTION

compf is a script file prepared to compile programs written in Fortran. *file_name* must be the name of a Fortran code file without extension.

compf will be found in directory */paros/bin* with the following contents:

```
#paros
\paros\bin\afserver.sn -:o 1 -:b \tf2v1\tf.b4 $1.f77 /I\paros\include
                    /I\tf2v1 /T8 /X /Q
```

Compilation on top of the Host OS

Considering DOS as the Host operating system, we will show the script files prepared to perform any compilation. You may change them, in order to adapt them to your needs.

NAME

ccomp

SYNOPSIS

ccomp *file_name*

DESCRIPTION

ccomp has been prepared to compile programs written in C. *file_name* must be the name of a C code file. No file extension must be specified.

ccomp.bat will be found in directory */paros/bin* with the following contents:

```
@echo off
IF "%1"==""  GOTO usage
IF NOT EXIST %1.c GOTO usage
\tc2v1\t8c %1.c /I\paros\include  /X /ZD
goto fin
:usage
echo Usage: %0 filename
:fin
echo Code compiled OK
```

NAME

fcomp

SYNOPSIS

fcomp *file_name*

DESCRIPTION

fcomp has been prepared to compile programs written in Fortran. **file_name** must be the name of a Fortran code file without extension.

fcomp.bat will be found in directory */paros/bin* with the following contents:

```
@echo off
IF "%1"==""  GOTO usage
IF NOT EXIST %1.f77 GOTO usage
\tf2v1\t8f %1.f77 /I\paros\include  /I\tf2v1 /X
goto fin
:usage
echo Usage: %0 filename
:fin
echo Code compiled OK
```

9.2.3 Linking

Within this version we also provide script files to link binary files with the corresponding libraries. As you will see in the script files, two steps are necessary to generate Paros executable files. The first step links all the binary files with the corresponding library, and the second step prepares the file so it can be loaded on top of Paros. In this step, the size of the application stack is specified (in integers). Paros executable files have the extension **.sn**. The intermediate files (those with extension .b4) may ne deleted, while not using *make* tool.

Linking on top of Paros

NAME

link

SYNOPSIS

link *file_name*

DESCRIPTION

link is a script file prepared to link binary files listed in *file_name*.ind. They will be linked with */paros/lib/user.lib* library. The executable file will be named after *file_name*.sn.

link will be found in directory */paros/bin* with the following contents:

```
#paros
\paros\bin\afserver.sn -:o 1 -:b \tc2v1\linkt.b4
          \transputer\lib\taskharn.t8+\paros\lib\user.lib+@$1.ind,$1.b4

\paros\bin\afserver.sn -:o 1 -:b \paros\bin\pclinker.a4 $1.b4 $1.sn 4000
```

link first links all the binary files listed in *file_name.ind* with /paros/lib/user.lib. It generates a file call *file_name.b4*. After that, the *pclinker* generates the executable file, named *file_name.sn* with a stacksize of 4000 integer. You may change the stack size if needed.

NAME

linkf

SYNOPSIS

linkf *file_name*

DESCRIPTION

linkf is a script file prepared to link binary files, listed in *file_name*.ind, with the fortran library */paros/lib/fortran.lib*. They will be also linked with */paros/lib/user.lib* library. The executable file will be named after *file_name*.sn.

linkf will be found in directory */paros/bin* with the following contents:

```
#paros
\paros\bin\afserver.sn -:o 1 -:b \tc2v1\linkt.b4
            \transputer\lib\taskharn.t8 + \paros\lib\user.lib +
            \paros\lib\fortran.lib+@$1.ind,$1.b4

\paros\bin\afserver.sn -:o 1 -:b \paros\bin\pclinker.a4 $1.b4 $1.sn 4000
```

Linking on top of the Host OS

We also provide some scripts to link on top of the Host OS. In the following descriptions we consider DOS as the host operating system.

NAME

clink

SYNOPSIS

clink *file_name1 file_name2 file_name3 ...*

DESCRIPTION

clink is a script file prepared to link up to 6 binary files. You may change it to your convenience. They will be linked with */paros/lib/user.lib* library. The executable file will be named after *file_name*.sn. **clink.bat** will be found in directory */paros/bin* with the following contents:

```
@echo off
IF "%1"==""  GOTO usage
for %%c in (%1 %2 %3 %4 %5 %6) DO IF NOT EXIST %%c.bin  goto usage
echo %1.bin  > %1.ind
IF EXIST %2.bin echo %2.bin >> %1.ind
IF EXIST %3.bin echo %3.bin >> %1.ind
IF EXIST %4.bin echo %4.bin >> %1.ind
IF EXIST %5.bin echo %5.bin >> %1.ind
IF EXIST %6.bin echo %6.bin >> %1.ind
linkt \tc2v1\taskharn.t8+\paros\lib\user.lib+@%1.ind,%1.b4
afserver -:b \paros\bin\pclinker.a4 %1.b4 %1.sn 4000
goto fin
:usage
echo Usage: %0 filename ...
:fin
:IF EXIST %1.b4 del %1.b4
:IF EXIST %1.bin del %1.bin
:IF EXIST %2.bin del %2.bin
:IF EXIST %3.bin del %3.bin
:IF EXIST %4.bin del %4.bin
:IF EXIST %5.bin del %5.bin
:IF EXIST %6.bin del %6.bin
```

```
:IF EXIST %1.ind del %1.ind
```

NAME

flink

SYNOPSIS

flink *file_name*

DESCRIPTION

flink is a script file prepared to link binary files, listed in **file_name**.ind, with the fortran library (/paros/lib/fortran.lib). They will also be linked with */paros/lib/user.lib*. The executable file will be named after *file_name*.sn.

flink.bat will be found in directory */paros/bin* with the following contents:

```
@echo off
IF "%1"==""  GOTO usage
for %%c in (%1 %2 %3 %4 %5 %6) DO IF NOT EXIST %%c.bin  goto usage
IF EXIST %1.bin echo %1.bin  > %1.ind
IF EXIST %2.bin echo %2.bin >> %1.ind
IF EXIST %3.bin echo %3.bin >> %1.ind
IF EXIST %4.bin echo %4.bin >> %1.ind
IF EXIST %5.bin echo %5.bin >> %1.ind
IF EXIST %6.bin echo %6.bin >> %1.ind
\tc2v1\linkt \tc2v1\taskharn.t8+\paros\lib\user.lib +
                              \paros\lib\fortran.lib+@%1.ind,%1.b4
afserver -:b \paros\bin\pclinker.a4 %1.b4 %1.sn 8000
goto fin
:usage
echo Usage: %0 filename ...
:fin
rem IF EXIST %1.b4 del %1.b4
rem IF EXIST %1.ind del %1.ind
```

NAME

linkalos

SYNOPSIS

linkalos *file_name1 file_name2 ...*

DESCRIPTION

linkalos is a tool that compiles and links C source files. It must be executed on top of the host operating system. In case you need to compile and link several files, it is more recommended to use *make* files, which distributes the compilations and linkings among different free processors, trying to do as many objectives in parallel as possible.

In case you want to prepare your own scripts files to compile and link, you have to keep in mind that the first input to the linker must be the *taskharn.t8* library. (look at *linkalos.bat* file in */paros/bin* directory). It is also necessary to pass the linker output through the PClinker program. The last parameter to this executable specifies the stack size to be reserved for each task.

9.2.4 Make files

NAME

make

SYNOPSIS

make [*-f makefile*] [*-i*] [*-j*] [*-n*] [*-p*] [*macro=val ...*] [*target ...*]

DESCRIPTION

make executes a list of commands associated with each target, typically to create or update a file of the same name. *makefile* contains entries that describe how to bring a target up to date with respect to those on which it depends, which are called dependencies. Since each dependency is a target, it may have dependencies of its own. Targets, dependencies, and subdependencies comprise a tree structure that **make** traces when deciding whether or not to rebuild a target.

make recursively checks each target against its dependencies, beginning with the first target entry in makefile. If after processing all of its dependencies, a target file is found either to be missing, or to be older than any of its dependencies, make rebuilds it. To build a given target, make executes the list of commands, called a rule.

In case, the option **-j** is used, make will check which objectives need to be executed and can be run in **parallel**. It starts as many objectives as possible in different processors.

Once an objective finishes, it checks if the upper objective can be executed (no dependencies left) and it that case executes it. This pattern is followed until the main targets are up to date.

When no makefile is specified with a -f option, make searches for a file named *makefile*, in the current working directory.

If no target is specified on the command line, make uses the first target defined in makefile.

OPTIONS

-f *makefile* Use the description file *makefile*.

-i Ignore error codes returned by commands.

-j Indicates *make* to execute as many objectives in **parallel** as possible, in different processors.

-n No execution mode. Print commands, but do not execute them.

-p Print out the complete set of macro definitions and target descriptions.

macro=value Macro definition. This definition overrides any regular definition for the specified macro within the makefile itself, or in the environment. However, this definition can still be overridden by conditional macro assignments.

USAGE

In this part, some examples of how *make* files look like on top of Paros will be shown. It is important to take into account which is the Host operating system, in order to specify correctly the path for the differents files.

Make with Unix as Host OS

```
CC = /paros/bin/afserver.sn -:o 1 -:b /tc2v1/tc.b4
CFLAGS = /I\paros\include /X /T8
LD = /paros/bin/afserver.sn -:o 1 -:b /tc2v1/linkt.b4
LD1 = \tc2v1\taskharn.t8
LD2 = \paros\lib\user.lib
PATCH = /paros/bin/afserver.sn -:b /paros/bin/pclinker.a4

distr.sn:       distr.b4
        $(PATCH) distr.b4 distr.sn 4000

distr.b4:       distr.bin rand.bin average.bin /paros/lib/user.lib
        $(LD) $(LD1)+$(LD2)+distr.bin+rand.bin+average.bin,distr.b4

rand.bin:       rand.c
        $(CC) rand.c $(CFLAGS)

distr.bin:      distr.c
        $(CC) distr.c $(CFLAGS)

average.bin:    average.c
        $(CC) average.c $(CFLAGS)
```

Note how the paths for the libraries are specified as in DOS. This is caused by the compiler and linker themselves. For them, slash "/" indicates that a new option is being introduced, so they can not be used to indicate paths. Files location must be specified with backslash.

Make with DOS as Host OS

Comparing with the previous makefile, it can easily be seen that the difference between them are in the path specification of the compiler, linker, libraries and the pclinker. In this case, all the path are specified in a DOS like syntax, while in the previous case, both DOS and Unix like syntax were needed.

```
CC = \paros\bin\afserver.sn -:o 1 -:b \tc2v1\tc.b4
CFLAGS = /I\paros\include /X /T8
LD = \paros\bin\afserver.sn -:o 1 -:b \tc2v1\linkt.b4
LD1 = \tc2v1\taskharn.t8
LD2 = \paros\lib\user.lib
PATCH = \paros\bin\afserver.sn -:b \paros\bin\pclinker.a4

distr.sn:       distr.b4
        $(PATCH) distr.b4 distr.sn 4000

distr.b4:       distr.bin rand.bin average.bin \paros\lib\user.lib
        $(LD) $(LD1)+$(LD2)+distr.bin+rand.bin+average.bin,distr.b4

rand.bin:       rand.c
        $(CC) rand.c $(CFLAGS)

distr.bin:      distr.c
        $(CC) distr.c $(CFLAGS)

average.bin:    average.c
        $(CC) average.c $(CFLAGS)
```

DIAGNOSTIC

If *make* don't know how to make targets it stops.

It is important to note that the commands that make can execute are those offered by Paros command line interpreter (both, built-in and non built-in commands).

A common error is to introduce blanks before the commands that *make* should execute. This causes error. There can only be one tabulator as in Unix makefiles.

9.2.5 Generating Paros configuration file with *cubata*

NAME

cubata (gin + coke)

SYNOPSIS

cubata *Inmos_conf_file Paros_conf_file*

DESCRIPTION

cubata utility reads a configuration file that specifies the logical structure of an application in a subset of the Inmos C Configuration Language and generates the configuration file used by the CSP loader provided with the Paros Free Distribution v1.2. *cubata* can be executed only on top of Paros environment. It is a tool provided in order to make easier the step of generating Paros configuration file, which may become tedious for some complex applications.

CONFIGURATION LANGUAGE SUMMARY

The configuration description offered by Inmos, consists of a software and a hardware network description. The aspects of the Inmos Configuration Language that are not considered are those very directly related to the physical hardware configuration. The description in the *Inmos_conf_file* file is considered a logical description of an application in terms of logical processors, tasks and channels. The following list is a summary of the main features of the Inmos Configuration Language supported by *cubata*:

Definitions

val *identifier exp*; Defines a numeric constant. The type is deduced from the type of the expression.

define *type* (attributes) *identifier*; Defines a node type. A list of attributes is optional.

Operators

Unary: + - !

Binary: + - * / % & — ¡¡ ¿¿ && —— ¡ ¿ ¡= ¿= == !=

Configuration types

connection A defined connection type between channels.

input Declares a software process *input* channel.

output Declares a software process *output* channel.

Language constructs

if *exp statement* [**else** *statement*]; Simple conditional construct. *exp* can be any valid integer expression and *statement* can be the single succeeding statement or a group of statements. **else** is optional.

rep *index* = *exp* **to** *exp statement*.
rep *index* = *exp* **for** *exp statement* Simple replication construct. Can be controlled by a range or a count.

connect *item* **to** *item* [**by** *connection*];
connect *item* **to** *item* **by** *connection*; Joins channels to channels. **by** is optional.

place *process* **on** *processor*; assigns a software process to a processor.

use *filename* **for** *process*; Assigns a linked unit to a process.

#include *filename* Includes another source file.

Node types

process (**interface** (input in, output out)) p; Each process possesses a set of attributes, but only the attribute *interface* are considered by *cubata*. This parameter is used to specify the list of channels that will be passed to each process. It corresponds to the line: Tn (Cn, Cm, ...) of Paros configuration file. Other parameters can not be passed to the process through this configuration file. The stacksize, heapsize and priority can not be specified from the configuration file. Parameters are passed from the command line interpreter to each task fo an application.

processor *pp[n]*; It is only used to define arrays of processors in order to *place* the processes on them.

USAGE

We will illustrate some examples in order to give a closer view of the typical configuration files accepted by the *cubata* utility in order to generate Paros configuration files. The next example can be found in directory /paros/user/tse:

```
val n 8;

/*{{{  Processor requirements */
processor (memory = 4M) m_processor[n];
/*}}}  */

/*  tasks and channels description */
process ( interface(input in, output out )) normal[n];

if (n>1)
{
 rep i = 0 to n-1
    {
      connect normal[i].out ,normal[(i+1)%n].in;
    }
}

/* tasks body and mapping to processor */
rep i = 0 to n-1
  {
    use "/paros/user/tse/tse.sn" for normal[i];
    place normal[i] on m_processor[i];
  }
```

From this Inmos configuration file, **cubata** produces the following configuration file to run on top of Paros:

```
Ptask description
number of processors :  8 logical
number of tasks :  8
number of channels :  8
number of executable files :  1

use /paros/user/tse/tse.sn in 8 task
   T7 in logical processor 7
   T6 in logical processor 6
   T5 in logical processor 5
   T4 in logical processor 4
   T3 in logical processor 3
```

```
   T2 in logical processor 2
   T1 in logical processor 1
   T0 in logical processor 0

declare channels
   C7 from T7 to T0
   C6 from T6 to T7
   C5 from T5 to T6
   C4 from T4 to T5
   C3 from T3 to T4
   C2 from T2 to T3
   C1 from T1 to T2
   C0 from T0 to T1

Task parameters
   T7 ( C6 C7 )
   T6 ( C5 C6 )
   T5 ( C4 C5 )
   T4 ( C3 C4 )
   T3 ( C2 C3 )
   T2 ( C1 C2 )
   T1 ( C0 C1 )
   T0 ( C7 C0 )
```

Another example would be the following configuration file:

```
val m 4;
val n (m-2);          /* number of processors */

/*{{{  Processor requirements */
processor (memory = 4M) m_processor;
/*}}}  */

/*  tasks and channels description */
process ( interface(input in, output out )) normal[n];
process ( interface(output out )) first;
process ( interface(input in, output lasto , input lasti)) last;

if (n>1)
{
 rep i = 0 to n-2
    {
      connect normal[i].out ,normal[(i+1)].in;
    }
}
connect normal[n-1].out, last.in;
connect first.out , normal[0].in;
connect last.lasto, last.lasti;

/* tasks body and mapping to processor */
rep i = 0 to n-1
  {
    use "/paros/user/migratio/bigplay.sn" for normal[i];
    place normal[i] on m_processor;
  }
use "/paros/user/migratio/bigplay.sn" for first;
place first on m_processor;
use "/paros/user/migratio/bigplay.sn" for last;
place last on m_processor;
```

which can be found in directory /paros/user/migratio. The paros configuration file obtained from the previous Inmos configuration file is:

```
Ptask description
number of processors :  1 logical
number of tasks :  4
number of channels :  4
number of executable files :  3

use /paros/user/migratio/bigplay.sn in 1 task
```

```
   T3 in logical processor 0

use /paros/user/migratio/bigplay.sn in 1 task
   T2 in logical processor 0

use /paros/user/migratio/bigplay.sn in 2 task
   T1 in logical processor 0
   T0 in logical processor 0

declare channels
   C3 from T3 to T3
   C2 from T2 to T0
   C1 from T1 to T3
   C0 from T0 to T1

Task parameters
   T3 ( C1 C3 C3 )
   T2 ( C2 )
   T1 ( C0 C1 )
   T0 ( C2 C0 )
```

9.2.6 SPMD *patch* utility

As explained in subsection 5.4.2, a SPMD loader has been included in this version of Paros Free Distribution. This loader creates at least as many tasks as physical processors not allocated with exclusion previously, but always between a minimum and a maximum value specified in the executable file.

In order to modify these values at will, we provide an utility called **patch**. This tool can also inform about the actual values of both bounds.

NAME

patch

SYNOPSIS

patch *file_name*.sn [*min_value*] [*max_value*]

patch *file_name*.sn ?

DESCRIPTION

patch is an utility that can be found in directory /paros/bin, and it can only be executed on top of Paros. It can show the actual values of the bounds set for an executable file, by typing the option **?**:

```
Paros_?patch tse.sn ?
Min = 1, and Max = 1
```

At link time, an application is considered to be sequential, so both bounds are set to one. After that moment, those values may be changed any time with no need of any recompilation of the application.

When executing the *patch* utility, the first value specified is considered to be the minimum amount of processors desired. If no maximum value is specified, it is set to -1, which means that the upper bound is set to the maximum number of physical processors that belongs to the network of transputers where Paros is running. If no value is specified, then the minimum value is set to 1 and the maximum to -1. If the first value is greater that the second, then both maximum and minimum values are to the greater number.

If both values specified are incoherent (0 or negatives values), both are set to 1. If only one value is specified and it is incoherent, the minimum is set to 1 and the maximum is set to -1.

USAGE

Here, we show some examples of how to use the patcher:

```
Paros_?link hello
Linking code...
Code linked OK
Paros_?patch hello.sn ?
Min = 1, and Max = 1
Paros_?patch hello.sn
Paros_?patch hello.sn ?
Min = 1, and Max = -1
Paros_?patch hello.sn 4 7
Paros_?patch hello.sn ?
Min = 4, and Max = 7
Paros_?hello.sn
hello world
hello world
hello world
hello world
hello world
hello world
hello world
Paros_?
```

STATUS

The actual loader gives priority to the CSP programming model. That means that if an application is loaded using a Paros configuration file, being the executable file modified by the patch utility, the loader will read the number of requested processors from the configuration file, rather that from the executable file itself.

9.3 Monitoring tools

NAME

mproc

SYNOPSIS

mproc [**-t***ticks*] [**-s***samples*] [**-w***bar width*] [**-m***means*] [**-p***proc_num*] [**-l**][**-d**]

DESCRIPTION

The *mproc* utility monitors CPU load for a certain amount of time and either displays the measures (through the VGA server) and/or writes them to the standard output.

For graphical output, *mproc* displays both kernel (green filled bar) and user (red filled bar) CPU load through the high resolution VGA server.

For file output, it lists lines indicating the processor number, percentage of time with user level threads running or ready to run and the percentage of time running threads at kernel level.

mproc is an executable file, found in directory /paros/user/load, that should be loaded in background before loading the application you want to monitor. The loader, as it supports the SPMD programming model, will try to load as many tasks as free physical processors are devoted to users applications. If this application is loaded with others still running, the kernel may group tasks thus disturbing measurements where more than one monitoring tasks are mapped and leaving un-monitored other processors.

OPTIONS

-t Determines the sampling interval in microseconds. The default value is 5000.

-s Determines the number of samples that will be used to estimate each mean load value. The default value is 100.

-m Determines the number of mean load values to be obtained. The default and maximum value is 500. It will record the number indicated, event in the case that the number of the mean values don't fin in the screen with the bar width specified. This allows a graphical view and a complete listing (if wanted, see next parameter) of the mean values.

-l List CPU loads to standard output. You can list the results event if you ask for screen display. In that case redirect the output to a file. If not, you will get the output on the screen together with the graphical display.

-w this parameter activates the graphical display and determines the width of the bars to be drawn. As default no graphical display is done. The minimum bar width is 1 and the maximum bar size depends upon the rest of the parameters. If the width specified is to big the monitoring application will complain. On the other side, *mproc* will only display the mean values that can fit into the screen with the bar width specified.

-d Delay before starting the sampling process (in 64 microsecond units).

-p *proc_num* Processor to be displayed or listed. Several processors can be selected. The maximum number allowed will be determined by the maximum number of processors in your system, with a limit of ten due to screen limitations. By default all processor's load are sampled, and the measures obtained are shown. This option is useful when the number of processors of your system is higher than 10, due to screen limitations.

USAGE

See section 6.3 and appendix B for examples of how to use it.

NAME

limon

SYNOPSIS

limon [**-l**] [**-m***means*] [**-s***samples*] [**-t***ticks*] [**-w***width*] [**-z**]

DESCRIPTION

The **limon** (load index monitoring)(lemon) utility continuously monitors CPU load of all the processors and displays it through the VGA server. **limon** displays both kernel (green filled bar) and user mode (red filled bar) CPU load.

limon is an executable file, found in directory /paros/user/load, that should be loaded in background before loading the application you want to monitor. The loader, as it supports the SPMD programming model, will try to load as many tasks as physical processors are devoted to users applications. If this application is loaded while others still run, the kernel may group tasks thus disturbing measurements where more than one monitoring task are mapped and leaving un-monitored other processors.

OPTIONS

-l List CPU loads to standard output. You can list the results event if you ask for screen display. In that case redirect the output to a file. If not, you will get the output on the screen together with the graphical display.

-m Specifies number of mean values that will be displayed and thus the duration of the continuous monitorization. The default value is 500. Very long periods can be monitored in this way.

-s Determines the number of samples that will be used to estimate each mean load value. The default value is 150.

-t Determines the sampling interval in microseconds. The default value is 5000.

-w This parameter activates graphical display and determines the width of bars that each processor will draw. In case the bars specified are too wide, limon tool will warn about it.

-z This option will force to continue monitoring, even after the specified number of mean values has been monitored, until the processor stays with zero load for 10 consecutive mean values.

USAGE

See appendix B for some examples of how to use it.

Appendix A

Installation Guide

This chapter describes the requirements and the installation procedure of the Paros Free Distribution v1.2.

A.1 Requirements

The hardware platform on which Paros Free Distribution v1.2 can be run is a network of at least three T800 processors. The processors must have 1Mbyte of memory or more. Paros Free Distribution v1.2 has been developed on a PC with a B008 board. This version of Paros has been targeted to PC hosted network. It also runs on UNIX or IDRIS V, with the absence of some features like graphics. Some of the scripts for booting, compiling or linking from the host have been adapted to UNIX or IDRIS scripts. In any case, they should be checked for local path names.

In order to improve the performance of the communication between Paros subsystems and the host server, and also to add new features to those subsystems, we have implemented our own host server, which is provided together with the rest of the system. Still, for application building the 3L C compiler and linker v2.1.1 (tc2v1) are needed. If you want to run applications with Fortran code, 3L Fortran compiler and linker (tf2v1) will also be needed.

One of the objectives of Paros Free Distribution is to be runnable on a large number of existing transputer platforms and on different transputer network configurations. For that reason Paros does not perform any configuration step. Before booting Paros, the network of transputers should be configured with the physical desired layout, using the tools provided by your distributor. Once the network is configured, you must generate the file */paros/etc/trans.cnf* which Paros will read to get the actual connections among the transputers that belong to the constructed network. The format in that file is that generated by the 3L utility **worm**, which will also be needed.

A.2 Contents

The Paros Free Distribution v1.2 can be obtained by anonymous ftp from

ftp.ac.upc.es

in directory /pub/soft/paros.

There you will find the following files:

paros_12.zip	: pkzipped file for the MS-DOS Paros Free Distribution v1.2.
paros_doc.ps	: postscript file for this documentation.
readme	

The actual Paros Free Distribution v1.2 comes in a single compacted file, which expands to the following set of directories:

paros/include	: *.h files for the user applications.
paros/lib	: user libraries
paros/bin	: scripts and executables to build an application and run the system (host OS executables)
paros/etc	: Executable files of the Paros Free Distribution Kernel v1.2 and subsystems
paros/user/example	: some examples commented in this manual
paros/user/comperf[1]	: example application to display communication speed
paros/user/load[1]	: monitoring applications
paros/user/migratio[1]	: application that migrates tasks
paros/user/mulmat	: application to solve matrix multiplication
paros/user/tse	: application to solve a triangular system of equations
paros/user/menu[1]	: example menu to execute previous applications
paros/user/fortran	: example of an application with fortran and C code
paros/user/queue[1]	: example that simulates a network of queues
paros/user/source	: source code of some heavy weight OS functions offered at subsystem level
paros/user/matrix	: source code of the example explained in the appendix
install.bat	:installation script
readme	:basic instructions to install and run PFD v1.2.

A.3 Installation

We now describe the installation procedure for DOS. For Idris V and Unix, refer to file README.INSTALL (which is provided in PFD v1.2 package).

[1]Not distributed on IDRIS V and Unix versions

Extract the packed files:

```
C:> mkdir c:\paros
C:> cd c:\paros
C:\PAROS> pkunzip -d -o paros_fd.zip
```

Append paths to environment:

```
C:\PAROS> set path=c:\paros\bin;c:\tc2v1;....
```

In order to install Paros you need a preconfigured network of transputers. You can fix your favorite topology although we recommend it to be as richly connected as possible.

Once you have configured it, type:

```
C:\PAROS> install
```

which runs the *worm* utility and based on its output configures the Paros Free Distribution Kernel v1.2 and subsystems. Recall that the first two processors found by the *worm* are not available for user applications. The *worm* writes the physical topology of the network to file /paros/etc/trans.cnf, which Paros reads when booting. The physical processor number of each transputer (see ps, allocate_physical_processor...) corresponds to the order in this file (the root processor is number 0). Processors 0 and 1 are reserved for system purposes, so user processors start at processor number 2. *Install* also runs the 3L command tools *config* which adapts the system to the memory size of the root processor.

For portability reasons and site organization, the paths needed by PFD v1.2 are specified in file /paros/etc/paros.env. So, in order to run Paros, you must specify the paths for the 3LC compiler, 3LC linker, taskharn.t8 library, your favourite editor, the host command line interpreter, the physical topology description file and the afserver executable file, prepared to run on top of Paros. Null string paths leads to system abort. Wrong specification in any path (except for the editor and the host cli) leads also to system abort. An example of environment file /paros/etc/paros.env for PC host OS (MS_DOS) is:

```
EDITOR   =   \bin\ne.com
SHELL    =   \dos\command.com
C\_3L     =  \tc2v1\tc.b4
LINKT    =   \tc2v1\linkt.b4
TASKHARN =   \tc2v1\taskharn.t8
CONF     =   \paros\etc\trans.cnf
AFSERVER =   \paros\bin\afserver.sn
```

This file can be changed in order to adapt it to your system.

Related to graphical displaying, there is an executable file in directory /paros/etc called graph.exe, which corresponds to a resident program that performs BIOS interrupt for VGA/CGA graphics and interacts with Paros vga/cga server. You must start this program before booting Paros. We recommend to add a line in autoexec.bat file that starts it:

```
\paros\etc\graph
```

You can now boot Paros Free Distribution v1.2 by typing:

```
C:\PAROS> Paros
```

In normal operation, the loader writes to file /paros/etc/paros.log a trace of the applications loaded and their terminations. The Kernel code has also a certain amount of trace messages in cases errors are detected. These messages are also written to file *paros.log*. We recommend that you look at this file in case something does not work as expected. You may execute the script file *sub.sh*, found in directory /paros/user/menu, which allows you to run some of the applications , prepared as representative examples of the programming style and functionalities supported by Paros.

We would appreciate if you report by e-mail to

paros@ac.upc.es

the problems you may detect.

Those users that work with a Root transputer with **1M**byte of memory must modify Paros launch script file (*\paros\bin\paros*). The option *CACHE0* must ne changed to *CACHEN*, resulting the following script file:

```
\paros\etc\hostser -:a -:b \paros\etc\main.app ROOT_OUT CACHEN
```

Appendix B

Parallelization of sequential programs

This appendix has been added with the objective of helpping to develop parallel programs on top of Paros. This appendix shows how a problem solved sequentially, can be distributed among different processes in order to be solved in parallel. We try to explain how to take profit of the flexible interface and utilities offered by Paros in order to access the maximum parallelism. We also show how monitoring tools may help to detect problems in the implementation.

The example choosen needs some numerical analisys background, but the main thing is not what the algorithm does, but how it solves the problem. The program calculates the ortonormalization of the columns of a matrix A^{mxn} by the Modified Gram-Schmidt method (MGS). The numerical parts of the code will not be described, our intention is not to teach numerical analisys.

B.1 Sequential solution

Let us start with the sequential version of the algorithm. This version solves the ortonormalization sequentially by a single process, that is a Ptask with one task. The application is loaded as a single executable file on top of Paros. The code for this example is listed bellow. You can observe how:

- the size of the problem to solve is passed to the application as arguments. Parameter

  ```
  -n
  ```

  corresponds to the number of rows and

`-m`

to the number of colummns.

- the memory is dynamically allocated. The *malloc_init* function must be previously called in order to initialize the heap.

You can execute this example by typing:

```
/paros/user/matrix/mgs_s.sn -n160 -m160
```

The code to be executed can be found in file *mgs_s.c*:

```
#include        "paros.h"
#include        "stdio.h"
#include        "sema.h"
#include        "misc_lib.h"
#include        "subsys.h"
#include        "math.h"

#define NR     1024
#define MC     1024
#define BUFFER 1024*4

typedef float fv[MC];

extern int errno;

void ini_data( nrows, mcolumms, vect)
int nrows,mcolumms;
fv *vect;
{
   int i,j;

   for (i=0; i<nrows; i++)
      for (j=0; j<mcolumms; j++)
         vect[i][j] = (i<j? 0.0 : 1.0);
}

void write_result( nrows, mcolumms, vect )
int nrows,mcolumms;
fv *vect;
{
   int i,j, max;
```

```
   max = ( nrows > mcolumms )  ? mcolumms : nrows;
   max = ( max > 10 ) ? 10 : max;

   for (i=0; i<max; i++)
   {
      printf("\t ");
      for (j=0; j<max; j++)  printf(" %1.0f", vect[i][j]);
      printf("\n");
   }
}


main(argc, argv, environ, n_channels, chan_list)
int argc;
char **argv;
char *environ;
int n_channels;
t_channelid ** chan_list;
{
   int nrows, mcolumms;
   int i, j, k, time;
   int print_on = 0;
   int myself;

   fv *vect;
   float *xnorm;
   float *sdot;


   nrows = 1024;
   mcolumms = 1024;

   get_task_index(&myself);


   for (i=0; i<argc; i++) {
      if (*argv[i] == '-'){
         switch (*(argv[i]+1)) {
            case 'n' : sscanf(argv[i]+2,"%d",&nrows);
                       break;
            case 'm' : sscanf(argv[i]+2,"%d",&mcolumms);
                       break;
            case 'p' : print_on = 1;
                       break;
            default  : break;
         }
```

```
    }
  }

  if (malloc_init((MC+2)*nrows*sizeof(float)+10000+BUFFER)== -1)
  {
    printf("ERROR in malloc_init \n");
    exit_terminate(1);
  }
  if ((vect = (float *)malloc(MC*nrows*sizeof(float))) == (float *)0)
    exit_terminate(1);
  if ((xnorm = (float *)malloc(nrows*sizeof(float))) == (float *)0)
    exit_terminate(1);
  if ((sdot = (float *)malloc(nrows*sizeof(float))) == (float *)0)
    exit_terminate(1);

  ini_data( nrows, mcolumms, vect);

  if ( print_on )
  {
    printf("\n\t ORIGINAL MATRIX \n\n");
    write_result(nrows, mcolumms, vect);
    printf("\n");
  }
  else
   printf(" TASK %2d : resolve from  0  to  %2d rows \n", myself, nrows-1);

  /* Normalization of the matrix "vect" */

  time = get_timer();
  for (i=0; i<nrows; i++)
  {
     xnorm[i] = 0.0;
     for (k=0; k<mcolumms; k++) xnorm[i] += vect[i][k]*vect[i][k];
     xnorm[i] = 1.0/sqrt(xnorm[i]);
     for (k=0; k<mcolumms; k++) vect[i][k]  = vect[i][k]*xnorm[i];
     for (j=i+1; j<nrows; j++)
     {
        sdot[j] = 0.0;
        for (k=0; k<mcolumms; k++)
        {
          sdot[j] += vect[i][k]*vect[j][k];
          vect[j][k] -= sdot[j]*vect[i][k];
        }
     }
  }
  time = (get_timer() - time)*64;   /* microseconds */
```

```
   if ( print_on )
   {
     printf("\n\t NORMALIZED MATRIX \n\n");
     write_result(nrows, mcolumms, vect);
     printf("\n\n");
   }
   printf(" SEQUENTIAL mgs  %d x %d  :  TIME = %d\n", nrows, mcolumms, time);
}
```

You will find the script *demol.sec* in directory */paros/user/matrix* that start the monitoring tool *limon* and this sequential version.

B.2 Parallel implementation

In this implementation, the normalization is solved distributing the matrix among several tasks. The distribution is interleaved by rows, that is, adjacent rows are distributed in different adjacent tasks. The code executed by each task is the same.

The number of tasks is decided when loading the application, either specifying it by a configuration file or by the SPMD programming model. That means that both, the size of the matrix and the number of tasks that will be started to solve the problem, are decided at load time. This is very useful to perform bentchmarks.

Note that the SPMD model can only be used when the code to be executed is the same in all the tasks of an application.

The tasks are configured as a logic ring, communicating with their neighbours by channels. The communication performed to get and distribute the final and intermediate row values has not been parallelized. The spread broadcast is linear in the ring.

This application can be loaded using the configuration file *mgs.4* . The code executed by each task will be found in file *mgs_p.c*:

```
#include "paros.h"
#include "stdio.h"
#include "sema.h"
#include "misc_lib.h"
#include "subsys.h"
#include "math.h"

#define NR     1024
#define MC     1024
```

```
#define BUFFER 1024*4

/* Initial set of vectors vect[nrows][mcolumms]  */
/* Distributed by (block,*)          */

typedef float fv[MC];

int myself;               /* my task number */
int mfr_gn;               /* my first row global number */

extern int errno;


void ini_data( lnr, mcolumms, vect)
int lnr,mcolumms;
fv *vect;
{
   int i,j;

   for (i=0; i<lnr; i++)
      for (j=0; j<mcolumms; j++)
         vect[i][j] = ((mfr_gn+i)<j? 0.0 : 1.0);
}

void write_result( lnr,mcolumms,vect )
int lnr, mcolumms;
fv *vect;
{
   int i,j, max;
   char buf[80], *buffer;

   max = ( lnr > mcolumms ) ? mcolumms : lnr;
   max = ( max > 10 ) ? 10 : max;

   buffer = malloc( (max*max*sizeof(float)) + (80*max));

   strcpy( buffer,"");
   for (i=0; i<max; i++)
   {
     sprintf( buf," T%2d : vect[%2d][0..%d] =",myself,mfr_gn+i,mcolumms-1);
     strcat( buffer, buf );
     for (j=0; j<max; j++)
     {
       sprintf( buf, " %1.0f", vect[i][j]);
       strcat( buffer, buf );
     }
     strcat( buffer, "\n");
```

```
   }
   printf( buffer );
   free( buffer );
}


void spread(buf, len, sender, num_tasks, east, west)
char *buf;
int  len;
int  sender;
int  num_tasks;
t_channelid east;
t_channelid west;
{
  int next;
  int error;

  len *= sizeof(float);

  if (sender == myself)
  {
   if ( error = channel_out(west, buf, len) )
    printf("task %d: error %d in channel_out : %x\n",myself, error, west);
  }
  else
  {
   if ( error = channel_in(east, buf, &len) )
     printf("task %d: error %d in channel_in : %x\n",myself, error, east);

   next = (myself + 1) % num_tasks;
   if (next !=sender)
    if ( error =  channel_out(west, buf, len) )
     printf("task %d: error %d in channel_out : %x\n", myself,error, west);
  }
}


main(argc, argv, environ, n_channels, chan_list)
int argc;
char **argv;
char *environ;
int n_channels;
t_channelid ** chan_list;
{
   int nrows,mcolumms;
   int lnr;      /* local number of rows */
   int global_i; /* index used to traverse the global number of rows */
```

```
int local_i;  /* local index */
int i,j,k;
int time;
int all_or_none;
int fr;
int sender;
int num_tasks;/* number of tasks */
int print_on = 0;

t_channelid east, west;

fv *vect;
float *xnorm;
float *sdot;
float *vbuf;
float *buff;

 /* get the logical configuration */

get_num_tasks(&num_tasks);
get_task_index(&myself);

 /* get channels identifiers from the main parameter list */

east = *chan_list[0];
west = *chan_list[1];

nrows = 1024;
mcolumms = 1024;

 /* get the rest of the parameters */

for (i=0; i<argc; i++) {
   if (*argv[i] == '-'){
      switch (*(argv[i]+1)) {
         case 'n' : sscanf(argv[i]+2,"%d",&nrows);
                    break;
         case 'm' : sscanf(argv[i]+2,"%d",&mcolumms);
                    break;
         case 'p' : print_on = 1;
                    break;
         default  : break;
      }
   }
}

lnr = nrows / num_tasks;   /* exact multiple for the moment */
```

```
   if (nrows % num_tasks != 0)
   {
      printf("ERROR: only exact multiple for the moment\n");
      exit_terminate(1);
   }
   mfr_gn = lnr * myself;

/* allocate memory space */

   if (malloc_init(MC*(lnr+3)*sizeof(float)+10000+BUFFER)== -1)
   {
     printf("ERROR in malloc_init \n");
     exit_terminate(1);
   }
   if ((vect = (fv *)malloc(MC*lnr*sizeof(float))) == (fv *)0)
     exit_terminate(1);
   if ((xnorm = (float *)malloc(MC*sizeof(float))) == (float *)0)
     exit_terminate(1);
   if ((sdot = (float *)malloc(MC*sizeof(float))) == (float *)0)
     exit_terminate(1);
   if ((buff = (float *)malloc(MC*sizeof(float))) == (float *)0)
     exit_terminate(1);

   ini_data( lnr, mcolumms,  vect );
   if ( print_on )
    write_result( lnr, mcolumms, vect);
   else
    printf(" TASK %2d : resolve from %2d to %2d rows \n",
                             myself, mfr_gn, mfr_gn+lnr-1);

     /* Normalization of the corresponding part of the matrix "vect" */

   time = get_timer();
   for (global_i=0; global_i<nrows; global_i++)
   {
      sender = global_i/lnr ;
      local_i = global_i % lnr;
      if (sender == myself)    /* global row is mine */
      {
         xnorm[local_i] = (float)0.0;
         for (k=0; k<mcolumms; k++)
            xnorm[local_i] += vect[local_i][k]*vect[local_i][k];
         xnorm[local_i] = (float)1.0/sqrt(xnorm[local_i]);
         for (k=0; k<mcolumms; k++)
            vect[local_i][k] *= xnorm[local_i];
      }
      if (sender == myself)
```

```
         vbuf = &vect[local_i][0];    /* broadcast normalized vector */
      else
         vbuf = buff;        /* use temporal buffer */

       /* do the broadcast */
      spread( vbuf, mcolumms, sender, num_tasks, east, west);

      all_or_none = (global_i<mfr_gn? 0 : lnr);
      fr = (sender==myself? local_i+1 : all_or_none);
      for (j=fr ; j<lnr; j++)       /* local numbering is enough */
      {
         sdot[j] = 0.0;
         for (k=0; k<mcolumms; k++)
         {
           sdot[j] += vbuf[k]*vect[j][k];
           vect[j][k] -= sdot[j]*vbuf[k];
         }
      }
   }
   time = (get_timer() - time)*64;   /* microseconds */

   if ( print_on )
     write_result( lnr, mcolumms, vect);

   printf(" TASK  %d: mgs %d x %d: time = %d\n", myself, nrows,
              mcolumms, time);
}
```

To run this example you only need to type:

```
/paros/user/matrix/mgs.4 -n160 -m160
```

You will found the script file *demol.par* that launches the monitoring tool *limon* together with parallel version of the Modified Gramm Schimdt algorithm.

B.3 Parallel loader version

In this version we have prepared a task (we could call it the *loader*) that loads the rest of the tasks that will perform the computations. We have prepared this example in order to show:

- how to load executable files at run time
- the use of some heavy weight functionalities provided by Paros libraries

- how to synchronize processes

This application is started as a sequential application with only one task that executes the code found in file *mgs_t.c*. This task loads the rest of the tasks which execute the code found in file *mgs_base.c*. The code executed by these tasks differs from the parallel version on the synchronization performed by the first ones, in order to get the timings correctly, and also on how to obtain the channel identifiers. In order to give more flexibility to this version, the number of tasks to be started is a parameter of the *loader* task (*[-t]*).

The main calls performed by this task are:

- task_create
- channel_create
- allocate_processors
- get_Ptask_cap
- lib_load_file_in_task
- Ptask_info

Refeer to sections7 and **??** to get a complete description of those calls.

The code executed by the *loader* task is:

```
#include "paros.h"
#include "stdio.h"
#include "sema.h"
#include "misc_lib.h"
#include        "subsys.h"
#include "math.h"

#define NR 1024
#define MC 1024
#define LOGICAL 0
#define SIZEBUF 1024

void  create_tasks( my_Ptask, task_table, proc_table, num_tasks )
t_Ptaskcap      my_Ptask;
t_task        *task_table;
t_logprocessorid  *proc_table;
int         num_tasks;
{
```

```
  int i, err = OK;

  for( i=0; i<num_tasks; i++)
  {
    task_table[i].tpt = proc_table[i];
    task_table[i].sl_size = 0;
  }
  if ((err = task_create ( my_Ptask, task_table, num_tasks)) != OK)
    printf("Error creating tasks \n");

}

void  create_channels( my_Ptask, task_table, chan_table, num_tasks )
t_Ptaskcap   my_Ptask;
t_task  *task_table;
t_channel *chan_table;
int num_tasks;
{
  int i, err = OK;
  t_taskid   iam, dest;

  for( i=0; i<num_tasks; i++)
  {
   iam  = i;
   dest = (i+1)%num_tasks;
   chan_table[i].source = task_table[iam].ident;
   chan_table[i].target = task_table[dest].ident;
  }
  if ((err = channel_create ( my_Ptask, chan_table, num_tasks))!= OK)
         printf("Error creating channels \n");
}

main( argc, argv )
int argc;
char **argv;
{
   int nrows,mcolumms;
   int i;
   int num_tasks  = 2;                   /* number of tasks */
   int print_on = 0;
   int nb_proc, size;

   t_channelid   east, west;
   t_task      *task_table;
   t_logprocessorid   *proc_table;
   t_channel *chan_table;
   t_Ptaskcap    my_Ptask;
```

```
t_tkinfo tk_info;
t_Ptkinfo Ptk_info;

char *arguments[6];
char buffer[6][12];
char bufprint[256], *printbuf;

static t_logprocessorid logprocessor[64];
static t_physprocessorid processor[64];
static t_segment segment[132];
static t_taskid task[64];
static t_channel channel[256];

get_Ptask_cap( &my_Ptask );

nrows = 1024;
mcolumms = 1024;

for (i=0; i<argc; i++) {
   if (*argv[i] == '-'){
      switch (*(argv[i]+1)) {
         case 'n' : sscanf(argv[i]+2,"%d",&nrows);
                    break;
         case 'm' : sscanf(argv[i]+2,"%d",&mcolumms);
                    break;
         case 'p' : print_on = 1;
                    break;
         case 't' : sscanf(argv[i]+2,"%d",&num_tasks);
                    break;
         default  : break;
      }
   }
}

/* memory space allocation for local tables */

size = num_tasks * (sizeof(t_task) + sizeof(t_logprocessorid) +
         sizeof(t_channel)) + SIZEBUF+1000;
if ( malloc_init(size) == -1)
{
  printf("ERROR in malloc_init \n");
  exit_terminate(1);
}
if ((task_table = (t_task *)malloc(num_tasks*sizeof(t_task))) == (t_task *)0)
  exit_terminate(1);
if ((proc_table = (t_logprocessorid *)malloc(num_tasks*sizeof(t_logprocessorid)))
 == (t_logprocessorid *)0)
```

```
    exit_terminate(1);
  if ((chan_table = (t_channel *)malloc(num_tasks*sizeof(t_channel)))
   == (t_channel *)0)
    exit_terminate(1);

  if (( printbuf = malloc(SIZEBUF) )== (char *)0)
    exit_terminate(1);


 nb_proc = num_tasks;
 if (allocate_processors(my_Ptask,&nb_proc,num_tasks,LOGICAL,
                          proc_table)!=OK)
 {
   printf("Error allocating processors\n");
   exit_terminate(1);
 }

 /* tasks creation */
 create_tasks( my_Ptask, task_table, proc_table, num_tasks );


 /* Channel creation */
 create_channels( my_Ptask, task_table, chan_table, num_tasks );

 /* Lanzar tasks */
 for (i=0; i<6; i++) arguments[i] = buffer[i];
 sprintf(buffer[0],"%d",nrows);
 sprintf(buffer[1],"%d",mcolumms);
 sprintf(buffer[3],"%d",num_tasks);

 for (i=0; i<num_tasks; i++)
 {
  east = chan_table[i].channel_id;
  west = (i==0)? chan_table[num_tasks-1].channel_id
                             : chan_table[ i-1].channel_id;

  sprintf(buffer[2],"%d",i);   /* logical task identifier */
  sprintf(buffer[5],"%d",east);
  sprintf(buffer[4],"%d",west);

  lib_load_file_in_task( "mgs_base.sn", 6, arguments, my_Ptask,
    task_table[i].ident, task_table[i].tpt );
 }

 if ( print_on )
 {
   clock_delay(1000);
```

```
 /*   example of getting information of a ptask ( like ps command ) */

Ptk_info.nb_physproc =   6;
Ptk_info.lpl_size   =   6;
Ptk_info.sl_size  =  32;
Ptk_info.til_size  =  32;
Ptk_info.chl_size  = 256;
Ptk_info.logprocessor_list = logprocessor;
Ptk_info.physproc_list     = processor;
Ptk_info.segment_list      = segment;
Ptk_info.taskid_list       = task;
Ptk_info.channel_list      = channel;
if ( Ptask_info( my_Ptask, &Ptk_info) != OK )
{
  printf (" WARNING: incomplet info on Ptask.\n");
  exit_terminate(1);
}

strcpy ( printbuf,"");
sprintf( bufprint,"\t                  PTASK   STATUS\n\n");
strcat ( printbuf, bufprint);
sprintf( bufprint,"\t:::::::::::::::::::::::::::::::::::::::::::::::::::\n");
strcat ( printbuf, bufprint);
sprintf(bufprint,"\tNumber of Physical Procesors:%d\n",Ptk_info.nb_physproc);
strcat ( printbuf, bufprint);
sprintf( bufprint,"\tNumber of Physical Procesors : %d\n",Ptk_info.lpl_size);
strcat ( printbuf, bufprint);
sprintf( bufprint,"\tRunning Tasks : %d\n",Ptk_info.til_size);
strcat ( printbuf, bufprint);
sprintf( bufprint,"\tChannels : %d\n",Ptk_info.chl_size);
strcat ( printbuf, bufprint);
sprintf( bufprint,"\tSegments : %d\n\n",Ptk_info.sl_size);
strcat ( printbuf, bufprint);
sprintf( bufprint,"\nPhysical processors list:");
strcat ( printbuf, bufprint);
for (i=0;i<Ptk_info.lpl_size;i++)
{
  sprintf( bufprint,"  %d",processor[i]);
  strcat( printbuf, bufprint);
}
strcat( printbuf, "\n\n");
sprintf( bufprint,"\nLogical processors list: ");
strcat( printbuf, bufprint);
for (i=0;i<Ptk_info.lpl_size;i++)
{
  sprintf( bufprint,"  %d",logprocessor[i]);
```

```
      strcat( printbuf, bufprint);
    }
    strcat( printbuf, "\n\n");

    sprintf( bufprint,"\nList of Tasks of Ptask:");
    strcat( printbuf, bufprint);
    for (i=0;i<Ptk_info.til_size;i++)
    {
      sprintf( bufprint,"  %d",task[i]);
      strcat( printbuf, bufprint);
    }
    strcat( printbuf, "\n\n");

    sprintf( bufprint,"\nChannels list:  ");
    strcat( printbuf, bufprint);
    for (i=0;i<Ptk_info.chl_size;i++)
    {
      sprintf( bufprint, "C%d (T%d,T%d) ",
               channel[i].channel_id,channel[i].source,channel[i].target);
      strcat( printbuf, bufprint);
    }
    strcat( printbuf, "\n\n");

    sprintf( bufprint,"\nPhysical segments list:\n");
    strcat( printbuf, bufprint);
    for (i=0;i<Ptk_info.sl_size;i++)
    {
      sprintf(bufprint,"\tSegment %2d: logical processor %2d, base %X,
               length %X\n", (int)segment[i].ident, segment[i].logicalp,
               segment[i].base,segment[i].len);
      strcat( printbuf, bufprint);
    }

    printf( printbuf );
  }
}
```

The code executed by the tasks that perform the analitical calculations is found in file *mgs_base.c* :

```
#include "paros.h"
#include "stdio.h"
#include "sema.h"
#include "misc_lib.h"
```

```
#include "subsys.h"
#include "math.h"

#define NR 1024
#define MC 1024

#define BUFFER 1024*4

/* Initial set of vectors vect[nrows][mcolumms]  */
/* Distributed by (block,*)          */

typedef float fv[MC];

int myself;              /* my task number */
int mfr_gn;              /* my first row global number */

extern int errno;


void ini_data( lnr, mcolumms, vect)
int lnr,mcolumms;
fv *vect;
{
   int i,j;

   for (i=0; i<lnr; i++)
      for (j=0; j<mcolumms; j++)
         vect[i][j] = ((mfr_gn+i)<j? 0.0 : 1.0);
}

void write_result( lnr,mcolumms,vect )
int lnr, mcolumms;
fv *vect;
{
   int i,j, max;
   char buf[80], *buffer;

   max = ( lnr > mcolumms ) ? mcolumms : lnr;
   max = ( max > 10 ) ? 10 : max;

   buffer = malloc( (max*max*sizeof(float)) + (80*max));

   strcpy( buffer,"");
   for (i=0; i<max; i++)
   {
     sprintf( buf," T%2d : vect[%2d][0..%d] =",myself,mfr_gn+i,mcolumms-1);
     strcat( buffer, buf );
```

```
     for (j=0; j<max; j++)
     {
       sprintf( buf, " %1.0f", vect[i][j]);
       strcat( buffer, buf );
     }
     strcat( buffer, "\n");
   }
   printf( buffer );
   free( buffer );
}
void syncronization( myself, num_tasks, east, west)
int myself;
int num_tasks;
t_channelid  east;
t_channelid  west;
{
  int size;
  int error;

  size= sizeof(int);

  if (myself == num_tasks-1)
  {
   if ( error = channel_out(west, (char *)&num_tasks, size) )
    printf("task %d: error %d in channel_out : %x\n",myself, error, west);
   if ( error = channel_in(east, (char *)&num_tasks, &size) )
     printf("task %d: error %d in channel_in : %x\n",myself, error, east);
  }
  else
  {
   if ( error = channel_in(east, (char *)&num_tasks, &size) )
     printf("task %d: error %d in channel_in : %x\n",myself, error, east);
   if ( error = channel_out(west, (char *)&num_tasks, size) )
    printf("task %d: error %d in channel_out : %x\n",myself, error, west);
  }
}


void spread(buf, len, sender, num_tasks, east, west)
char *buf;
int  len;
int  sender;
int  num_tasks;
t_channelid east;
t_channelid west;
{
  int next;
```

```
  int error;

  len *= sizeof(float);

  if (sender == myself)
  {
   if ( error = channel_out(west, buf, len) )
    printf("task %d: error %d in channel_out : %x\n",myself, error, west);
  }
  else
  {
   if ( error = channel_in(east, buf, &len) )
     printf("task %d: error %d in channel_in : %x\n",myself, error, east);

   next = (myself + 1) % num_tasks;
   if (next !=sender)
    if ( error =  channel_out(west, buf, len) )
     printf("task %d: error %d in channel_out : %x\n", myself,error, west);
  }
}


main( argc, argv )
int argc;
char **argv;
{
   int nrows,mcolumms;
   int lnr;  /* local number of rows */
   int global_i;   /* global row index */
   int local_i;   /* local row index */
   int i,j,k;
   int time;
   int all_or_none;
   int fr;
   int sender;
   int num_tasks;                    /* number of tasks */
   int print_on = 0;

   t_channelid east, west;

   fv *vect;
   float *xnorm;
   float *sdot;
   float *vbuf;
   float *buff;

   if ( argc < 6 ) printf("ERROR me faltan argumentos\n");
```

```
i=0;
sscanf(argv[i++],"%d",&nrows);
sscanf(argv[i++],"%d",&mcolumms);
sscanf(argv[i++],"%d",&myself);
sscanf(argv[i++],"%d",&num_tasks);
sscanf(argv[i++],"%d",&east);
sscanf(argv[i++],"%d",&west);


lnr = nrows / num_tasks;   /* number of rows to be treated by this task,
                              exact multiple for the moment */
if (nrows % num_tasks != 0)
{
   printf("ERROR: only exact multiple for the moment\n");
   exit_terminate(1);
}

mfr_gn = lnr * myself;

             /* memory allocation */

if (malloc_init(MC*(lnr+3)*sizeof(float)+10000+BUFFER)== -1)
{
  printf("ERROR in malloc_init \n");
  exit_terminate(1);
}
if ((vect = (fv *)malloc(MC*lnr*sizeof(float))) == (fv *)0)
  exit_terminate(1);
if ((xnorm = (float *)malloc(MC*sizeof(float))) == (float *)0)
  exit_terminate(1);
if ((sdot = (float *)malloc(MC*sizeof(float))) == (float *)0)
  exit_terminate(1);
if ((buff = (float *)malloc(MC*sizeof(float))) == (float *)0)
  exit_terminate(1);

ini_data( lnr, mcolumms,  vect );
if ( print_on )
 write_result( lnr, mcolumms, vect);
else
 printf(" TASK %2d : resolve from %2d to %2d rows \n",
   myself, mfr_gn, mfr_gn+lnr-1);

syncronization ( myself, num_tasks, east, west);

 /* Normalization of the corresponding part of matrix "vect" */
```

```
   time = get_timer();
   for (global_i=0; global_i<nrows; global_i++)
   {
      sender = global_i/lnr ;
      local_i = global_i % lnr;
      if (sender == myself)    /* global row is mine */
      {
         xnorm[local_i] = 0.0;
         for (k=0; k<mcolumms; k++)
            xnorm[local_i] += vect[local_i][k]*vect[local_i][k];
         xnorm[local_i] = 1.0/sqrt(xnorm[local_i]);
         for (k=0; k<mcolumms; k++)
            vect[local_i][k] *= xnorm[local_i];
      }
      if (sender == myself)
         vbuf = &vect[local_i][0];    /* broadcast normalized vector */
      else
         vbuf = buff;        /* use temporal buffer */

        /* do the broadcast */
      spread( vbuf, mcolumms, sender, num_tasks, east, west);

      all_or_none = (global_i<mfr_gn? 0 : lnr);
      fr = (sender==myself? local_i+1 : all_or_none);
      for (j=fr ; j<lnr; j++)       /* local numbering is enough */
      {
         sdot[j] = 0.0;
         for (k=0; k<mcolumms; k++)
         {
           sdot[j] += vbuf[k]*vect[j][k];
           vect[j][k] -= sdot[j]*vbuf[k];
         }
      }

   }
   time = (get_timer() - time)*64;   /* microseconds */

   if ( print_on )
     write_result( lnr, mcolumms, vect);

   printf(" TASK  %d: mgs %d x %d: time = %d\n", myself, nrows, mcolumms, time);
}
```

In order to run this example you have to type:

```
/paros/user/matrix/mgs_t.sn -n160 -m160 -t4
```

The parameter *-t4* indicates the number of tasks that will be loaded in parallel.

You will find scripts files that starts the monitoring tools (*limon* and *mproc*) for each version presented. There is also a makefile prepared to generate all the implementations. Try to run the makefile in parallel.

Appendix C

GLOSSARY

Capability Identifier for global kernel supported objects. Capabilities are typed and contain the encrypted access rights on the object. The following list defines the access right on each type of object: Kernel: Owner, Cluster_create and Raw_allocate Cluster: Owner, Cluster_resize, Ptask_create and Info Ptask: Owner, Ptask_Manage and Segment Port : Owner, Read and Write

Channel Synchronous point to point communication mechanism between threads within a Ptask. Provides the very fast communication support needed for parallel programming.

Cluster Processor space protection domain. It also has functions of resource management control. Ptasks are created within one cluster.

From the protection point of view in a transputer based system, only Ptasks that are confident on each other should be created in the same cluster.

From the processor management point of view, Ptasks in the same cluster may share processors (so the full power of the processor will not be available for each Ptask). It also provides a quota mechanism to allow subsystems to control the resources that a user program can directly request to the kernel.

Exception port Port associated to each Ptask. On detection of a hardware exception the kernel sends a message to this port indicating the cause and thread state of the causing thread.

Logical processor Identifier that allows the user to specify which tasks should be grouped in a single physical processor. The mapping of logical to physical processors can be controlled by the user (requesting one to one mapping) or left to the kernel.

Physical processor Processor with local memory.

Physical processor identifier Each processor is identified by a fixed processor number

Port Synchronous and flexible communication mechanism between threads in a single or different Ptasks. Provides and n to n communication support in a system wide basis. It supports system oriented communications (rather that parallel programming) such as access to servers,...

Processor quota Maximum number of physical processors that can be allocated on the whole by all the Ptasks running in a given cluster.

Ptask Encapsulation of a parallel program and the resources allocated to it in order to be run.

Segment Contiguous sections of memory in the physical address space of one processor that has been allocated to a Ptask. It can be mapped into the logical address space of one task.

Task Each of the separate logical address spaces within a Ptask.

Thread Sequential flow of execution within a task

Thread state The information that is necessary to restart (through the thread_create primitive) a thread that has previously been suspended (through the Ptask_suspend or task_suspend primitive)

Index