The paper analyses the genesis and the rationale of the idea of multithread processes, and some of the related issues:

- support of different granularities of parallelism,
- explicit vs. implicit intra-threads synchronization,
- kernel vs. user support of threads,
- relations with the object oriented paradigm,
- relations with Remote Procedure Calls (RPC) and distributed systems,
- relations with open systems.

The paper focuses in particular on the application of multithread processes in real-time systems, and on the way parallelism can be better exploited in these systems.

Processes, Task Forces and Threads.

The main goal of multitasking operating systems (OS) is to allow the specification of cooperating sequential processes that can execute concurrently: the logical parallelism allowed by concurrency may become actual parallel execution in case the OS manages a multiprocessor machine. This computational model is supported by almost all OSs, both multiuser (e.g. Unix [20]) and real-time oriented (e.g. VRTX [19], pSOS [18]). The system is constructed as a set of processes.

In traditional OSs a process consists of a single address space and a single thread of control within the address space. The execution of a program within a process entails maintaining a great deal of state information, such as register (and program counter) values, page tables, swap image, file descriptors, ... In a multiuser OS like Unix the volume of the state information is large, and makes processes very expensive to create and maintain [2]; this is not true in real-time executives where the process state information is quite simple (typically only registers and the program counter) [21] [18]. Moreover, in real-time OSs all processes of a single processor share the same address space, thus processes are not enclosed in separate, protected virtual address spaces. Process are scheduled for execution of their thread and allocated resources independently from each other. Cooperation is supported by the OS providing implicit or explicit mechanisms to synchronize the execution and the exchange of information between processes.

Real-time OSs are a special type of small-kernel OSs. In the small-kernel approach the executive provides only the basic services of process management and interprocess communication, whilst all functions of the OS are implemented by server processes. Small-kernel OSs have borrowed, for local interprocess communications, models typical of distributed computing environments (i.e. message passing). Separate components of the OS can be placed in disjoint domains (address spaces), with message passing mechanism used for interdomain communications. Modular structure, and easy design and maintenance are some of the advantages of this approach [4][27].

The experience has shown that, for modularity and performance, a single service can be better provided by a set of cooperating activities rather than by a single sequential process [7] [24]. In order to facilitate the interaction of these activities several systems provide some form of support of the notion of task force. A task force is a set of concurrent activities that cooperate closely in the execution of a single logical task. Activities are the entities that are actually scheduled for execution on processors: depending on the implementation, activities may be full processes [7] [18] or similar to processes except that they exist only as part of a task force [10] [16]. The main characteristic of a task force is to allow a finer grain of interaction between its members, by special forms of synchronization (e.g. implicit synchronization through scheduling rules) and/or of sharing of the execution environment (address space, resources allocated, ...).

This notion leads to the definition of 2 levels (or granularities) of parallelism, internal to the task force, and external, between task forces.

A form of task force that is widely discussed is that of multithread processes [24] [10] [5] [6] [8]. In such systems, a process (the task force) consists of an address space, and one or more threads of control. Each thread has its own program counter, registers file and stack, but all threads share the address space and other state informations.

In multiuser systems a significant difference between processes and threads lays in the efficiency of the management operations: creation, deletion, context switch (e.g. in Unix the creation of a thread is 10 times cheaper than the creation of a process [24]). This is however not necessarily true (in MESA the cost of creating or destroying a process is of the same order of that of a procedure call [12] - but in MESA all processes share the same address space), and in particular is not true in real-time OSs, where processes are light-weight anyway, so that any efficiency gains allowed by threads would be minimal [21][18].

Relations with RPCs and the Object Paradigm.

Although threads represent a general purpose mechanism to provide concurrency within a single execution environment [24] [16] [18], from the very beginning they have been associated to more specific programming paradigms, such as the server and the object models [10] [5].

Moreover, threads provide a nice solution to many of the problems caused by blocking RPC. A server can run calls from multiple clients concurrently, each in a separate thread. Similarly, a client can create several threads to perform work concurrently:
Threads, in conjunction with RPC, represent therefore also a software engineering tool, since they simplify the structuring of subsystems. Consider for instance the implementation of layer N in a layered network architecture. Each request from layer N to layer N+1 may be associated with a different thread, created upon arrival of the request (an RPC): the thread may block when issuing requests (RPCs) to the N-1 layer, without blocking the whole layer N, and maintaining in its stack state information about the partial execution of the request: the RPC mechanism guarantees that the response returned from layer N-1 is given to the thread that actually issued the request. A thread can then be destroyed after confirming the service to the requestor (return of the RPC). This minimizes housekeeping operations that would be necessary with an implementation based on processes and messages. Other examples of this idea are presented in [14] (a file system) and [24].

Notice that in these examples a service can be modeled as an object, a service request via an RPC as the invocation of an operation on the object, and a thread is associated with each operation execution (a pool of precreated anonymous threads might wait for incoming requests in order to improve efficiency): so we see a synergy between the ideas of threads, RPC, and object oriented programming [3] [5] [8].

Implementation Issues and Relations with Open Systems

MACH [1] provides a support of multithread processes directly within the kernel, but the same concept can be implemented also at user level in a standard Unix environment, where the OS kernel manages only single thread processes [8] [23] [25]: a library is linked to the user program in the user space, to support creation, destruction, scheduling and synchronization of threads.

The 2 solutions are fundamentally different: the first one, where threads are directly implemented by the OS kernel, requires a special purpose support, while the second one can be implemented on any existing OS, thus allowing a multithread programming environment to be uniformly provided as the basis for open distributed processing (ODP) [8].

Another advantage of the user space solution is that it allows a more flexible definition of threads management policies:

- threads may be precreated and made dormant while not servicing user request, rather than being created and destroyed;
- scheduling may be preemptive or not, with or without priorities, and so on.

The user level implementation presents also some disadvantages [24], which however must be carefully examined. All these problems are related to the fact that the native OS kernel considers a process to be single threaded.

- It is impossible to fully exploit a multiprocessor architecture, making different threads of a same process simultaneously run on different processor. Although the statement is correct, this is probably not a major problem: the computational power can still be used by other processes, exploiting the external parallelism of the system. In fact even OS kernels that directly support several granularities of parallelism in multiprocessor architectures constrain all threads of a process to be executed by a same processor [22]. There are other architectural considerations that decrease the relevance of this constraint in real-time system, that will be discussed in the next sections, as well as its positive side.
- If a thread executes a blocking system call of the native OS, all threads of the process are blocked. This is true only in poor user level implementations. RPCs are now efficient enough to represent a suitable universal OS interface [4]; so it is enough that all OS services are provided via this interface, and that this interface is implemented using nonblocking primitives, to guarantee that the problem will not arise. In fact this is the approach taken in ODP oriented implementations, where the threads support library is actually a full "local execution environment" which interfaces and indirectly provides all OS services, in a form which is independent from the particular, native OS [8].

In the user implementation a single thread can also block the whole process because it generates a page fault. There is no way to overcome this problem except for locking the whole address space of the process (or parts of it under user control) in memory. Although this is not a general solution, one should consider the following points:

- the possibility of locking in memory the address space of a process is considered a necessary extension to Unix in order to support real-time applications [17];
- in most real-time OSs there is no support of virtual memory, and the whole address space of the system is locked in memory [18] [19];
- from a more general standpoint, the block of a process should not be considered a disastrous event because external parallelism guarantees that the system as a whole will not be blocked. In fact this is what happens in single threaded systems when a process takes a page fault: it stops providing its service until the page is available in memory.

So we can conclude that the introduction of a user level multithread support package in a Unix-type environment provides all the software engineering advantages of threads, is suitable for the construction of open systems, and does not imply any worsening of the present single thread situation. Moreover, in environments aimed at the support of real-time systems, it can be also as efficient as a kernel implementation.

Work done at Washington University shows that user level threads in conjunction with shared memory can also be used (very efficiently) for distributed programming, both in loosely and in strictly coupled multiprocessor environment [2] [26]. However a strong distinction may arise between small objects, volatile and internal to the distributed program, and large object, permanent and external to the program: it is not obvious that both objects can be acted upon through the same RPC support [2] [26].

Synchronization

Because threads are independently scheduled, the system must provide some synchronization facility, although not necessarily via explicit primitives.

Mutual exclusion, for instance, can be provided implicitly by the scheduler if this is non-preemptive and threads cannot run simultaneously on different processors in a multiprocessor architecture: this is always the case for a user implementation of threads in a Unix-type environment. This strategy is widely adopted in real-time systems, and for the construction of task forces [7] [18] [19], and it allows to transform a constraint of user level implementations into an overall efficiency gain. The constraint on the scheduling policy is also not severe, since all threads of a process cooperate to support a single logical task, so that there is no reason to implement timesharing or priority based scheduling among them. The efficiency gain affects not only the user program but also the threads support library, whose procedures need not
be explicitly synchronized (they can be programmed as if there were no critical regions within the library).

In this hypothesis no explicit synchronization primitives are needed except for the simple sleep/wakeup pair, since interthread communication is supported in the most efficient way by the shared data space of the process.

Of course, different approaches may be taken, as in MACH [24] where the kernel allows several threads of a process to be simultaneously executed on different processors, and explicit thread synchronization primitives are supported in a multiprocessor environment. While this approach is general and efficient, it is also very complex, and not suitable as the basis for ODP because it is not compatible with existing OSs.

**Parallelism in Real-Time Systems**

Most real-time applications (in robotics, automation, telecontrol and telecommunications [21] [15]) are built using hierarchical system architectures, even when the physical structure of the hardware platform is made of peer processors connected via a N*N communication structure (e.g. a memory bus in a shared memory multiprocessor). Each node is often functionally specialized (e.g. because of the peripheral resources connected), so that it is convenient that it is equipped with a local OS tailored to the requirements of the functions that it hosts (e.g. different types and degrees of real-time constraints, need for a file system or a high level human interface). Thus, it appears that in real-time systems it is not convenient to use a single, monoprocessor OS, but rather a set of heterogeneous monoprocessor OSs, capable to interoperate with each other in an open environment over several types of interconnections.

The strictly coupled (shared memory) approach seems to be discouraged even by the most advanced commercial memory busses [11], where the primary communication mechanism is message passing. This suggests that the global functionality be provided through the cooperation of independent local entities. In these systems a kernel implementation of multithread processes would not provide any additional actual parallelism, but would prevent the construction of an open distributed environment of heterogeneous OSs.

The previous discussion has shown that all disadvantages of the user level implementation of threads need not be present in a real-time environment where the address space of a process is locked in memory. This applies in particular to the support of multithread processes in real-time Unix-like environments, which often represent the upper levels in the hierarchical structure of real-time systems.

When we come to computational nodes with higher real-time requirements different OSs must be considered [18] [19] even if we can assume that they are all based on the concurrent sequential processes model like Unix (this is not strictly true now, especially in very hard real-time applications, see for instance [15]). However, the applicability of the model is being extended by the raw power of the new microprocessors. Its advantages in terms of system engineering and modularity over more primitive approaches make it the most desirable choice [21]).

All these OSs support single thread processes, but these processes are themselves very light weight: for instance pSOS+ and VRX perform a context switch in 20 to 80 microseconds, a Motorola 68020 at 25 MHz. Therefore it is unlikely that in this case threads could be supported with higher efficiency than processes. On the other hand the more general idea of task force is still applicable and convenient: in some ways it is often also supported by these OSs (e.g. through implicit mutual exclusion based on scheduling rules).

Even though we have just argued that there is no reason to support multithread processes in this context, this does not mean that one cannot apply the design methodologies that have been developed from the synergy of the ideas of multithread processes, RPC, and object oriented programming. In this case a task force can be associated to each object, an RPC to an operation request to an object, and a single thread process to the service (execution) of each operation request.

The extensions necessary for the support of this abstraction can be provided by a user level library, without the need of any specific kernel support: this library could also take care of lower level optimization such as the management of a pool of anonymous precreated processes. Notice that there is no substantial difference between this library and the one that implements at the user level multithread processes in a Unix-like environment, and in fact they should provide the same application interface if we aim at the construction of a real-time ODP environment.

At this moment there are no evidences that this approach is actually viable. On the contrary, experimental measures performed in a small-kernel, real-time environment, suggests that the management of processes is still too costly to allow a fine granularity for objects implemented as task forces (the approach tends also to limit the use of large shared data structures). The RPC related parallel objects paradigm appears to be applicable, but at two disjoint levels:

- large grain objects, implementable as kernel managed task forces;
- fine grain objects, all part of a same program, where operations can be implemented as user level threads, just as in a Unix-like environment.

**A case of study: a communication subsystem**

An example of the programming model allowed by typical real-time executive is given by an Inter Process Communication (IPC) facility developed in Telettra for real-time distributed applications (for telecommunications and telecontrol). These systems are based on a proprietary, throughput oriented, real-time, small kernel that supports a shared address space for all processes, and the concept of task force through implicit mutual exclusion.

The IPC subsystem, like all the other OS services, is implemented outside the kernel, by means of dedicated server processes. It has the layered architecture shown in figure 1. The IPC interface is somewhat similar to Unix BSD sockets [13].

Each layer is implemented as a task force. For example the Logical Link Control (LLC) layer is a task force with four processes. A first process acts as the server of the user processes: it receives the requests and the responses from the users (i.e. open and close of connections, transmission of messages) and passes these requests to the Medium Access Control (MAC) layer, without waiting for responses. The second process receives the responses from the MAC layer and passes them to the user (e.g. the IPC layer). The third process acts as the server of the indications posted by the MAC layer, and passes the messages received to the user. The last process manages the timers required by the connection oriented IEEE 802.2 protocol. This four processes cooperate efficiently by means of shared data structures and without explicit synchronization. Communication between layers is accomplished by means of the kernel supported local message exchange primitives.
null server procedure, according to the model shown in figure 3, implies two context switches, takes only 405 microsec. A special instance of the more general abstraction of task force. A slow operation (a cross domain local RPC with context switch) with the LRPC package on a C-VAX workstation takes 157 microseconds [4].

This architecture has many advantages in terms of modularity, ease of implementation and maintenance, but has serious performance drawbacks. In spite of the efficiency of the local message passing primitives of the executive, the implementation of each single layer as a separate task force introduces a great overhead that substantially reduces the global performance of the communication subsystem (independently of the type of service offered by the layers, e.g. connection oriented vs. connectionless services in LLC). For example the introduction of the IPC interface layer and of a connectionless LLC service over the MAC layer, reduces of more than 50% the throughput (messages per second) of the LAN (figure 2 and [9], which examines this point in more detail). A more integrated implementation appears convenient, as suggested also by the Berkeley IPC [13], and by the examples provided in [4]. For example, it would be possible to insert all layers of the IPC service in a single task force.

<table>
<thead>
<tr>
<th>MAC user</th>
<th>314</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPC user</td>
<td>148</td>
</tr>
<tr>
<td>- connectionless</td>
<td></td>
</tr>
<tr>
<td>- without copy</td>
<td></td>
</tr>
</tbody>
</table>

Test Condition:
- Throughput = messages per second over the specified interface
- Half Duplex Traffic
- MAC SDU length = 31 byte

Fig. 2 - Throughput of the IPC service

The viability of the small-kernel approach is shown by the efficiency that can be achieved by a local RPC emulated via the message exchange support. The efficiency can be tested using a local null RPC implemen-
tationless services in the executive, the implemen-
tation of each single layer

Finally, it has been recognized that there exists a strong synergy between the idea of task force and those of RPC and object oriented programming, and that this fact has emerged thanks to development of thread based systems: these 3 ideas are likely going to provide the framework from which an ODP environment (suitable also for real-time applications) will be developed. However, a substantial gap remains between large grain and fine grain objects: this fact seems to require different implementation strategies. In real-time systems, multiple threads, implemented at user level, appear best suited to support fine grain, parallel objects, all part of a same program.

References


