North-Holland
Microprocessing and Microprogramming 27 (1989) 251–258

HARDWARE SUPPORT FOR THE TUMULT REAL-TIME SCHEDULER

H.C. van der Bij, G.J.M. Smit, P.J.M. Havinga.

Twente University of Technology
Dept. of Computer Science
P.O. Box 217
7500AE Enschede, The Netherlands

Keywords: real-time, scheduling, multi-processors, scheduling hardware.

Abstract

This article decribes the hardware which is designed for speeding up and supporting the schedule routines of the TUMULT multi-tasking operating system. TUMULT uses a "priority running up" schedule algorithm which automatically increases the priority of a process when (part of) it must be finished before another process, with a higher priority, can run. Also the theoretical backgrounds of the schedule algorithm and properties for some classes of schedule problems are described.

1. Introduction

The use of computers for control and monitoring of industrial processes expanded dramatically in recent years and will probably expand even more in near future. A computer used in such an applications is often shared between time-critical control and monitor functions and some non-time-critical tasks. To fulfil the demands of the time-critical tasks and to still use the computer efficiently, careful scheduling of processes must be done.
Already in the early 70's, theories were developed on scheduling processes with timing constraints. Most of the simpler theories are still being used. However, more elaborate algorithms, which also take into account the dependancy of processes, are hardly ever used. This is probabably caused by the unfamiliarity with the disadvantages of the simple algorithms and because of the extra processing time needed by the more complex algorithms.
At the Department of Computer Science of the Twente University, research is being done on both hardware and software aspects of a real-time multi-processor system called TUMULT. We have sped up the implementation of the schedule algorithm used by TUMULT by designing dedicated hardware [Bij 88]. We have also collected some theoretical backgrounds of the algorithm. This article describes some of the results of this research.

2. Overview of TUMULT

TUMULT (Twente University MULTi-processor system) is a modular expandable real-time multi-processor (MIMD) system [Jansen 88]. All memory is distributed and nodes communicate via message passing. The system is intended for high performance real-time applications such as robot control, pattern recognition, gateways, etc.

The hardware consists of up to 64 nodes, that communicate via a high performance switching network (bandwidth 20 Mbytes/sec.). Each node consists of a Network Interface (NI) with a Communication Processor (CP) and one or more Host processor(s) each with individual, local memory (see fig. 1).

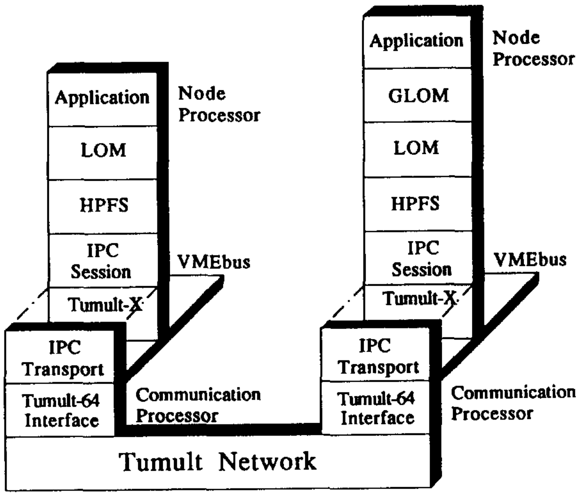


Fig. 1. The hardware architecture of TUMULT.

The Network Interface provides a flexible and reliable message passing service to the Host Processor(s). The communication protocol for this service is handled by the Communication Processor [Smit 88]. The Network Interface is connected to the Host processor(s) of that node via a VMEbus interface. The Host Processor(s) executes the distributed operating system as well as the application program.

The operating system is written in Modula-2 [Wirth 85] and comprises the following layers [Jansen 88]:

1. Kernel.
A real-time multi-tasking kernel called TUMULT-X [Luttmer 89] runs at each node. It offers primitives such as memory allocation, process creation, process termination, scheduling, interrupt handling and exception handling.

2. Inter-Process-Communication (IPC).
This layer allows for dynamic creation and deletion of logical communication links. A link is a flexible communication structure that can be adapted to the current communication needs of the system.

3. High Performance File System (HPFS).
It allows files and devices to be distributed over the nodes transparently [Langen 87]. It inherits the dynamic behaviour from the IPC primitives.

4. Local and Global manager (LOM and GLOM).
These layers receive, interpret and execute user commands and collect status information of the allocated processes.

3. Scheduling hard-real-time processes

Multi-tasking operating systems allow users to structure their applications as a set of communicating concurrent tasks. Tasks are initiated by external events (such as interrupts) or by other tasks.

For real-time applications, like process control, in general tasks must be completed before a certain fixed point in time, called the deadline. If a system requires a service within a fixed time, this environment is called "hard-real-time", in contrast to "soft-real-time", where a statistical distribution of response time is acceptable.

The mechanism that determines which task is to be executed at a particular moment is called the scheduler. There are basically two scheduling algorithms. One is called **non-deterministic** scheduling, where the arrival times of the processes are not known in advance. The other is called **deterministic** scheduling where the moment when a process will get processor time is determined before all processes are started. In the latter case the scheduler has to know in advance which processes are available, their start and execution times and their deadlines.

There is an extensive list of literature on theory of deterministic scheduling [see Liu 74]. The most important problem of these theories is that the parameters of the tasks (starting time, execution time, deadline etc.) have to be known in advance. For our intended applications where processes are created dynamically and/or processes are initiated by external events such as interrupts, this condition does not hold.

Non-deterministic scheduling algorithms (like the TUMULT-X scheduler) are based on the assignment of priorities to processes. The scheduler takes care of switching in the highest priority process which is runable.

Most real-time schedulers are of the preemptive type, which means that a running task will be interrupted whenever a task with a higher priority is initiated or unblocked.

There are three ways of assigning priorities to processes:
- static priority: priorities are assigned to processes once and for all.
- dynamic priority: the priority may change from request to request.
- mixed priority: some processes have static priority and others have dynamic priority.

Liu and Layland [Liu 73] showed that for dynamic priority assignments the deadline scheduling is optimal. Optimal in the sense that no other scheduling algorithm can schedule tasks that cannot be scheduled by the deadline scheduling algorithm.
It assigns priorities to processes according to the deadline of their current requests. If the deadline of the current request is near (or far away), a high (or low) priority will be assigned to the process.
Unfortunately Liu's theory is only valid for repetitive and mutual independent processes.
The constraint of repetition was later released by Labetoulle [Lab 74]. However, the constraint of mutual independency will generally not be met because processes often share resources, such as printers, disks, memories and I/O devices. Therefore, the early theories on scheduling, which assume that the tasks are independently running and are competing for processor time only, cannot be used anymore.

Consequence of process dependency

The dependency of processes gives rise to special problems. The following example shows that by introducing dependency, a process with a high priority can be blocked by lower priority process (see fig. 2).

Example 1.

> Suppose a low priority task (P1) has claimed a resource, which can be used by one process at a time. Next, while the resource is claimed, a medium priority process (P2) is started. P1 then will be preempted because P2 has a higher priority. After that, a high priority process (P3) is started, which also needs the resource claimed by P1.
> Because the resource is claimed, P3 will be blocked until P1 has released the resource. However P1 cannot continue because it was preempted by the medium priority task (P2). The result is that the medium priority process blocks the high priority process because the low priority process cannot release the resource.

Another problem is that it is very difficult to assign priorities to tasks that can be made runable by several other processes. In practice a lot of fine-tuning is required to obtain optimal performance.

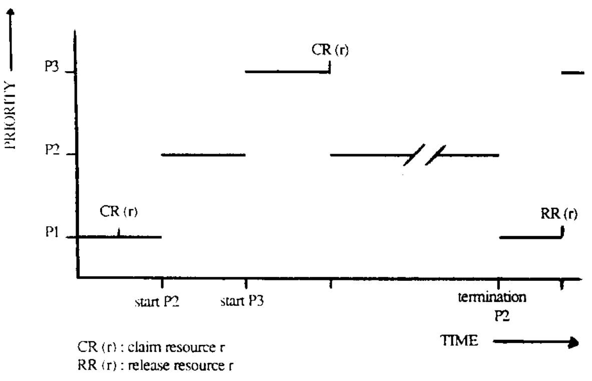


Fig. 2. The high priority process will be blocked by a medium priority process if the simple algortihm is used.

The running up algorithm

To overcome the above-mentioned problems, a "running up" schedule algorithm is employed. In this paragraph only the basic outline of this algorithm is given. The algorithm will be presented more detailed in the next section.

Basically a running up scheduler tries to allocate all processor time to the process with the highest priority. It does so by trying to free the resources needed by the process with the highest priority. This can be accomplished by (temporary) incrementing the priority of the processes that have claimed the resources which are needed by the highest priority process. If the priority is incremented up to that of the process which needs the resource, the process which has claimed the resource will be switched in by the scheduler. If the needed resource is released, the high priority process will continue and the low priority process will return to its low priority.

Example 2.

Same task-set as in example 1: P3 needs a resources that is claimed by a low priority process (P1). In the "running up" algorithm P1 will be switched in, instead of P2. P1 temporarily gets the same priority as P3. When P1 releases the resource, P3 will continue and P1 restores its old priority.

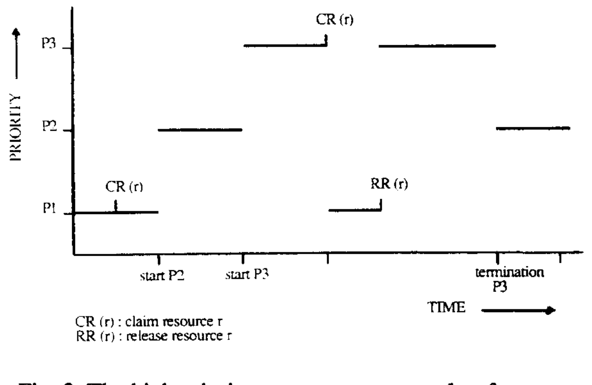


Fig. 3. The high priority process can proceed as fast as possible if the running up algorithm is used (cf. fig. 2).

In 1976 Blazewicz presented and proved a deterministic, optimal algorithm for scheduling dependent tasks with different arrival times [Blaz 76]. His algorithm is comparable with our running up algorithm, but it has the severe disadvantage that the arrival times and deadlines of all tasks must be known in advance. As stated before this is not true in general for a real-time executive, where processes can be created and deleted dynamically.

Example 3 shows that no schedule algorithm can be optimal if the arrival times of processes, the precedence constraints (process dependencies) and deadlines are not known in advance.

Example 3.

Two task sets and their only possible schedules are given. Recall that the scheduler does not know what will happen in the future. The only thing it knows are the deadlines of the tasks already started and the resources claimed during the execution of these tasks. At time T=3 the scheduler has to run P2 in case a, while in case b P1 must continue and the execution of P2 must be postponed. At T=3 the scheduler cannot distinguish between case a and case b because it cannot look into the future.

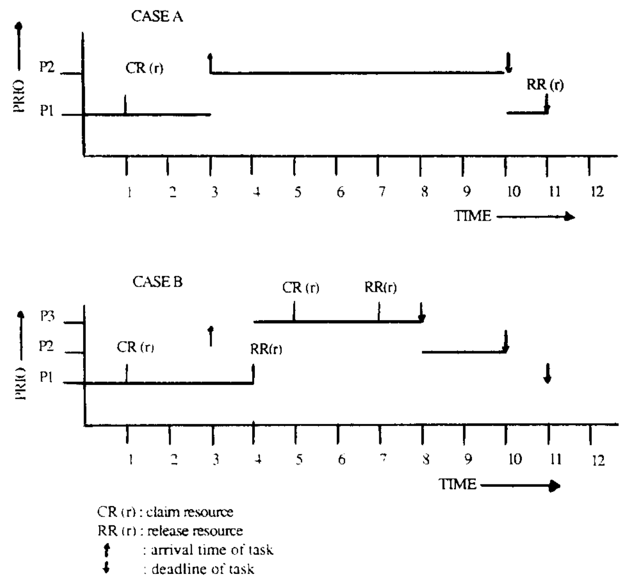


Fig. 4. The problem of non-deterministic scheduling.

This example shows that no scheduler can always schedule in an optimal way if the parameters of the processes are not known in advance.

4. The Tumult executive

Introduction

The TUMULT executive is the part of the kernel software that takes care of the process administration, scheduling, context switching and synchronization.

The TUMULT executive uses a hard-real-time deadline scheduling algorithm. It tries to allocate resources (processor time, memory, devices etc.) in such a way that the processes terminate before their respective deadlines. The executive assigns priorities to processes dynamically, such that processes with the earliest deadline have the highest priority. It uses a "running up" mechanism (as described in section 3) to avoid the drawbacks of a common priority system.

It is proven that the Tumult running up scheduler will schedule processes in an optimal way in two cases [Bij 88]:

1. The processes run independently. In this case the algorithm will just schedule the task with the highest priority and because a deadline priority assignment is used the algorithm then is optimal (see section 3).
2. The processes get a priority according to their deadline, the precedence constraints are set at the start of the processes and all processes start at the same time. If these conditions are valid, the running up algorithm will schedule the task-set in an optimal way, because it then corresponds with the Blazewicz algorithm, which is proven to be optimal.

As shown in section 3, no deterministic schedule algorithm can be optimal if the processes do not start at the same time, so it appears that the Tumult running up scheduling algorithm has good properties.

For process synchronization the executive uses a primitive datatype called 'SYNC', with which semaphores, mutual exclusion for resources, events etc. are implemented. If a process is not waiting for a SYNC, but waiting for the processor, then we say that it is waiting for the "processor SYNC".

We first describe the scheduling algorithm.

The scheduling algorithm

All living processes are placed in a list which is sorted in priority order. This list is called the **schedule list**. Every process in it points to the synchronization primitive where it is waiting for. The scheduler inspects this list to find a process that can be switched in, i.e. a process that is waiting for a "processor SYNC". The priority of the tasks is assigned according to their deadlines: Pj := Dj, where Pj denotes the priority of task j and Dj its deadline. Note that a low value implies a high priority.

There are two routines, "WaitSync" and "SignalSync" which operate on syncs and their associated counters and queues. WaitSync decrements the counter if it is not 0 and blocks a process if the counter value is 0. SignalSync increments the counter and can unblock a process. Syncs are not different from Dijkstra semaphores [Dijkstra 68].

Specification of the algorithm

Informally the "running up" schedule algorithm can be described as follows:

> If a medium priority process M can unblock a sync for which a higher priority process H is waiting, then temporarily increase the priority of process M to the priority of H until it releases the sync. We call this new priority the **effective priority** of M. If process M is waiting for a sync blocked by another process L, then give process L also the priority of H, etc. Assign the processor to the process with the highest effective priority.

More formally the running up algorithm can be specified as:

Step 1

For every task Tj in the schedule list T1..Tn determine

$$EP_j = \min \{ P_j, \min_{i \in [1..n]} (P_i \mid T_j < T_i) \}$$

where EP_j denotes the effective priority of task Tj, Pj denotes the priority of task j, Tj < Ti means that task j can signal a sync by which task i is blocked.

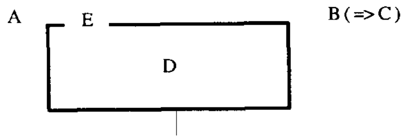
Step 2

Assign the processor to the runable task Tj, which has the minimum value EP_j. Process it until either it is completed, it is blocked or another process Tk with a lower effective priority arrives or is unblocked.

Repeat these two steps until all tasks are scheduled.

5. The implementation of the running up algortihm

To explain the used scheduler implementation, the schedule list is visualized in a diagram. Each rectangle represents a process in the schedule list.



A: priority number
B: sync a process is waiting for
C: number of process that can signal sync B
D: list of syncs that can be signalled
E: process number

Fig. 5. A process in the schedule list.

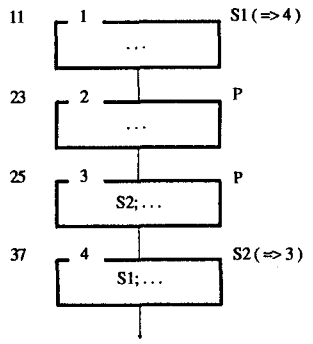


Fig. 6. Example of a schedule list.

In this example the algorithm will select process 3. Process 1 is blocked by semaphore S1 which can be signalled by process 4. In its turn, process 4 is blocked by semaphore S3 which can be signalled by process 3, which is not blocked and runable. When process 3 releases S2, its effective priority will fall back from 11 to 25 and process 4 is switched in.

Next the software implementation of the algorithm is given.

Step 1
Take the first process from the schedule list and the SYNC for which it is waiting. This process is called the hunter process HH.

Step 2
Inspect the SYNC: if it is not blocked goto 4, else goto 3.

Step 3
If the SYNC has a signaller, then take that process as a new hunter and goto 2. If the SYNC has no signaller, take the first process after HH as the new hunter HH. Goto 2

Step 4
Switch this process in.

Notes:
- The algorithm shown is not optimized in any way.
- In the current implementation it is not possible that a SYNC can be signalled by more than one process.
- In the current implementation a process can be blocked by only one SYNC at a time.

Because this algorithm is rather complex and because the executive is called quite often it was assumed that it uses a significant part of the available processor time. Therefore we started a project to investigate whether dedicated hardware for supporting the time-consuming functions of the executive would be profitable or not.

6. The schedule hardware

The hardware scheduler implements the above described scheduling algorithm. It strongly resembles the software implementation, but works independently of it and in parallel with the Host processor. The hardware determines according to the running up algorithm which process has to get processor time. Furthermore an associative memory is included which is able to find the position where a new process has to be inserted in the (in priority order sorted) process list.
The basis of the design is formed by 3 memory modules that have the following functions (see fig. 7):

1. Linked Priority List (LPL memory)
   It contains the link pointers of the processes which are sorted in priority order.
2. Process List (PL memory)
   It contains the number of the SYNC (e.g. resource) a process is waiting for.
3. SYNC List (SL memory)
   This memory contains the number of the process which can signal a blocked SYNC. It contains two extra bits which indicates whether a SYNC is blocked or not and whether the signaller is known or not.

The scheduler works as follows:

Step 1.
First the number of the process with the highest priority, called the Hunter-Process is read from LPL memory location 0 and fed to the HunterRegister

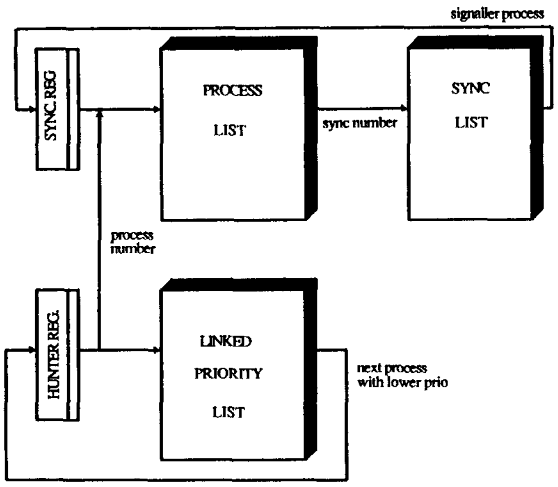


Fig. 7. Block diagram of the hardware scheduler.

(HR). The content of the HR register is fed as an address to the PL memory.

Step 2.
The PL memory provides the number of the SYNC this process is waiting for. Next this SYNC number is fed as an address to the SL memory.

Step 3.
The content of the SL memory indicates whether this process is runable or not. If the SYNC is blocked and the signaller process of this SYNC is known (which is indicated by the two extra bits), the SL memory provides the number of the signaller process.
This process number is fed back via the synchronization register to the PL memory. Steps 2 and 3 are repeated until a process is found which is either runable or a SYNC is found which is blocked and has no explicit signaller.

Step 4.
If a runable process is found, this process will be assigned to the processor. If no runable process is found because the list came to a dead end (the SYNC cannot be released), a new Hunter-Process (i.e. the process with the next lower priority) must be found. This new Hunter-Process can be found in the LPL memory, since this memory contains a linked list of processes in priority order. The above algorithm continues until a process is found which is runable.

Searching the position where a process has to be linked into the priority order sorted process list can also take a lot of time. Therefore this function is implemented in the design as well. It required only one additional comparator to the design of fig. 7 (see fig 8). The priority of each process is allocated in the PL memory at the same location as the pointer to the SYNC. If the priority is requested (indicated by an extra address bit) the priority of the process will appear at the output. The comparator is connected to the data output of the PL memory. The priority of the process that must be inserted in the list is fed to the other input.

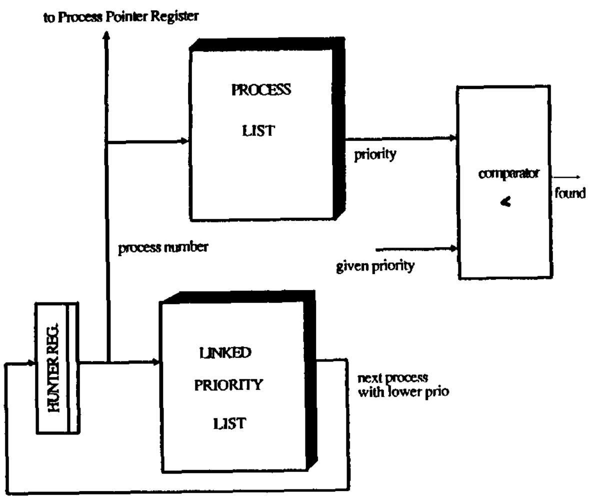


Fig. 8. Inserting a new process in the Linked Priority List.

The hardware passes through the Linked Priority List, continuously monitoring the priority of the processes. At some time it will encounter a process with a priority lower than the priority of the given process. The new process will be inserted just before this one. The number of its predecessor can be found in the Process Pointer Register. Basically this design implements a content addressable memory.

Realization

A prototype of the hardware scheduler has been built and preliminary test results have been obtained.

The current realisation can store up to 255 processes and 4095 Syncs. In practice these limitations are not very restrictive. The prototype is built with off the shelf components, such as EPLDs, PALs and fast memory chips. The circuit has the size of a single Eurocard.

The design possibly can be integrated. Approximately 7 Kbytes of memory is required for holding 256 processes, 4K Syncs and a priority range of 2^{32}.

7. Performance

The circuit is tested with 35 nsec RAMs. The clock frequency of the prototype is 25 MHz.

For a realistic comparison it is assumed that the software implementation runs on a MC68000 running on 16MHz with zero wait-states. The software implementation uses some clever optimizations to the described basic algorithm. It will be clear that some assumptions have to be made about the complexity of the schedule list. For example if the first process to be checked is runable, both hardware and software scheduler will find a runable process very fast. In the calculations for the hardware version the use of parallelism of the hardware and the software is not taken into account, because in general not many instructions can be executed while the hardware is busy. However it can speed up the functions by some microseconds. Table 1 shows some results for different schedule lists.

In this list we assume that only the last process to check is runable.

Number of processes	Software [µsec]	Hardware [µsec]	Notes
10	47	2	SYNC without signaller
100	384	17	SYNC without signaller
10	219	4	Running up lists of 1 process
100	2030	34	Running up lists of 1 process
10	389	11	Running up lists of 2 processes
100	3652	101	Running up lists of 2 processes

Table 1. Performance results of the hardware scheduler (one schedule session).

One can see that the hardware version is much faster if there are a lot of processes that have to be checked. At first glance this result may seem very promising. However, it is expected that the lists to be checked are short, even if many processes are started.
The reason for this is that processes with a high priority have the earliest deadline and if it is not possible to make these runable they probably will not meet their deadline.

Another reason why in total not much speed up can be expected, is that programmers are well aware of the fact that the overhead of process switches is large and so they avoid them whenever possible. In addition to that, when programs are split up into processes they will try to keep the dependency of processes low, so the running up list will be short.

If we assume that:
1. half of the number of releases of a resource unblocks another process
2. 2/3 of the released resources unblock a process with a higher priority,

then about 9 to 230 µsec can be won by using the hardware for every WaitSync and SignalSync pair. If for example 200 Wait/Signal operations are performed per second, then 0.2 to 4.6 % of the available processor time can be won. However because of the above mentioned reasons, it is most likely that only the low end of this speed-up range will be achieved

8. Conclusion and results.

The running up algorithm as used in Tumult appears to have good properties for hard-real-time scheduling. It is proven to be optimal if the processes are independent and also if the processes start at the same time and have precedence constraints.
A prototype of the hardware scheduler has been built and preliminary performance results were obtained.
The obtained speed-up of the hardware is strongly dependent on the complexity of the schedule list, the number of allocated processes, the number of Syncs and of course on the total number of times the scheduler is called. The first tests showed that a hardware scheduling operation takes 20 to 40 µsec. Without the hardware support it is estimated that it can vary from 30 to 600 µsec assuming that at most 15 processes in the process lists have to be checked. The benefit of the hardware scheduler thus strongly depends on the application. Because of this, more tests with real applications have to be performed to validate the extra cost of a hardware scheduler.

Acknowledgements

We gratefully acknowledge the staff members of the University of Twente, The Dr Neher Laboratory and Océ Nederland B.V. as well as the numerous students without whose support the implementation would never have been possible. Especially we thank Maurice Luttmer for the fruitful discussions on real-time scheduling.

References

[Bij 88]: Bij H.C.van der; "Hardware for the Tumult hard-real-time scheduler", M.Sc. Thesis, Twente University of Technology, Dept. of Computer Science, June 1988.

[Blaz 76]: Blazewicz J.; "Scheduling dependent tasks with different arrival times to meet deadlines", Modelling and performance evaluation of computer systems, pg 57-65, 1976.

[Dert 74]: Dertouzos, M.L.; "Control Robotics: the procedural control of physical processes", Proc. IFIP congress 1974.

[Dijkstra 68]: Dijkstra E.W.; "The structure of the T.H.E. multiprogramming system", Communications of the ACM, pg 341-346, 1968.

[Henn 78]: Henn, R.; "Antwortzeitgesteuerte Processorzuteilung unter strengen zeitbedingungen", Computing, Vol 19. pg 209-220, 1978.

[Jansen 88]: Jansen P.G., Smit G.J.M.; "Tumult-64: a real-time multi-processor system", int. rep. nr INF-88-18, Twente University of Technology, 1988.

[Lab 74]: Labetoulle, J.; "Ordonnancement des processus temps reel sur une ressource preemprive", Thess de 3eme cycle, Universite Paris VI, 1974.

[Langen 87]: Langen P. van, Sijbers A.: "The distributed file system in Tumult", Conference on computing science in the Netherlands, Amsterdam, 1987.

[Liu 73]: Liu C.L., Layland J.W.; "Scheduling algorithms for multiprogramming in hard real-time environment", Journal of the ACM, 20(1), pg. 46-61, Jan. 1973.

[Liu 74]: Liu C.L.; "Deterministic job scheduling in computing systems", Modelling and performance evaluation of computer systems, pg. 241-253, 1974.

[Luttmer 89]: Luttmer M.L.M., Jansen P.G.; "TUMULT-X: A real-time executive"; int. rep., Twente University of Technology, 1989.

[Smit 88]: Smit G.J.M., Jansen P.G.; "The communication processor of Tumult-64", Proceedings Euromicro 88, pg 519-524, Zurich, 1988.

[Wirth 85]: Wirth N.; "Programming in Modula-2", Third corrected edition, Springer Verlag, 1985.