

A Timeliness-Guaranteed Kernel Model - DREAM Kernel - and Implementation Techniques

K. H. (Kane) Kim ⁺
University of California, Irvine, U.S.A.

Luiz Bacellar ⁺

Jungguk Kim
HUFS, Korea

Yuseok Kim ⁺

and

Chittur Subbaraman ⁺

Kee-Wook Rim
ETRI, Korea

Hankil Yoon ⁺

ABSTRACT: An essential building-block for construction of future real-time computer systems (RTCS's) is a *timeliness-guaranteed operating system*. The first co-author recently formulated a model of an operating system kernel which can support both real-time processes and new-style real-time objects with guaranteed timely services. The model has been named the DREAM kernel. The key emphasis in formulating the DREAM kernel was in realization of guaranteed timely service capabilities with minimal loss of hardware utilization. This paper presents a summary of the main structuring principles that were exploited to realize guaranteed timely service capabilities together with modularity and expandability in the DREAM kernel. A prototype implementation of the DREAM kernel, v.D2, has been produced by the authors to run on a network of PC's connected by an Ethernet. Several implementation techniques that were adopted during the course of this prototype implementation and may be applicable to other real-time kernel development environments, are briefly discussed in this paper. The prototype kernel (v.D2) has been used to run a real-time object structured non-trivial defense C3 application together with a real-time simulator of the application environment.

1. Introduction

A few years ago the first co-author started subscribing to the view that it was time to start vigorously pursuing the following real-time computing paradigms [Kim94c, Kim95b]:

- (1) *General-form design style*: Future real-time computing must be realized in the form of a generalization of the non-real-time computing, rather than in a form looking like an esoteric specialization.
- (2) *Design-time guarantee of timely service capabilities of subsystems*: To meet the demands of the general public on the assured reliability of future RTCS's in safety-critical applications, there does not appear to be any adequate way but to require the system engineer to

produce design-time guarantees for timely service capabilities of various subsystems (which will take the form of objects in object-oriented system designs).

The motivating factors behind these paradigms which may be called the new-generation (NG) real-time computing paradigms are the newly improved hardware economy and component reliability which provide impetus in expanding the real-time computing application field.

An essential building-block for construction of NG real-time computer systems (RTCS's) is a *timeliness-guaranteed operating system* which together with the hardware platform forms an *execution engine* that provides guaranteed timely services to concurrent and distributed real-time application software. If the operating system lacks such guaranteed timely service capabilities, then timely service capabilities of real-time applications cannot be ensured. Existing commercial operating systems lack the capability for either supporting a general-form design style or providing guaranteed timely services to application software.

The first co-author recently formulated a model of an operating system kernel which can support real-time processes with guaranteed timely services. The model, named the DREAM (Distributed Real-time Ever Available Micro-computing) kernel, actually came out of an attempt to develop an execution engine that supports a new real-time object-oriented (OO-) structuring approach. The new object structuring approach is called the RTO.k object structuring scheme [Kim94a, Kim94b, Kim94c] and intended to facilitate the NG real-time computing. To realize the idealistic NG real-time computing, a powerful structuring scheme capable of dealing with all practically useful real-time and non-real-time computing requirements must be established. We believe that such structuring methods should preferably be of object-oriented (OO-) type, considering the modularity, generality, and natural abstraction benefits that object-oriented approaches bring in. In the last several years, there has been a growing trend of research activities aimed for extending the conventional OO-structuring approaches to support RTCS design [Att91, Ish92, Kim94b, Kop90, Tak92]. However, most of those works have not been aimed for supporting the design-time guarantee of timely service capabilities of objects, which is one of the fundamental requirements of the NG

⁺ These co-authors are staff members of the DREAM Laboratory, ECE Dept., Univ. of California, Irvine (UCI), USA, directed by the first co-author.

real-time computing. Therefore, the RTO.k object structuring scheme, though an extension of the conventional OO-structuring approaches, has several unique features of fundamental nature which will be reviewed later in Section 3.1.

As an execution for RTO.k objects, the DREAM kernel was designed to enable flexible linking between RTO.k objects and various hardware structures. The kernel enables this by supporting

- (1) real-time processes with various activation and synchronization requirements,
- (2) shared data structure monitors, and
- (3) real-time multicast logical (RML-) channels, which in turn execute the components of RTO.k objects.

Therefore, the DREAM kernel can support both process-structured real-time application software and RTO.k object structured application software. The key emphasis in formulating the DREAM kernel was in realization of guaranteed timely service capabilities with minimal loss of hardware utilization. The DREAM kernel can thus be viewed as a model of a *general-purpose timeliness-guaranteed OS kernel*.

One of the two main purposes of this paper is to provide an overview of the main structuring principles that were exploited to realize guaranteed timely service capabilities together with modularity and expandability in the DREAM kernel. Among the main principles exploited is that of structuring multiple layers of “time-leasing” machines, each of which guarantees a certain amount of machine time being available to upper-layer machines. The DREAM kernel has a five-layer structure and the lower four of the five layers form the DREAM micro-kernel.

A prototype implementation of the DREAM kernel, v.D2, has been produced by the authors to run on a network of PC's connected by an Ethernet. Presenting some major implementation techniques adopted is the other main purpose of this paper. The prototype kernel (v.D2) has been used to run an RTO.k structured non-trivial defense C3 application, together with a real-time simulator of the application environment.

The paper starts in Section 2 with a discussion on the core of the DREAM kernel that is essentially a timeliness-guaranteed process execution engine. The entire DREAM kernel is then discussed in Section 3 and this section begins with a review of the essence of the RTO.k object structuring scheme. The remainder of the section focuses mainly on the part the DREAM kernel that is directly related to supporting RTO.k objects. Section 4 discusses the prototype implementation, v.D2, and some implementation techniques adopted are presented. The paper concludes in Section 5 with discussions on the issues to be resolved via future research.

2. DREAM kernel as a process execution engine

The DREAM supports both process-structured real-time application software and RTO.k object structured application software. This section focuses on the core of the DREAM kernel that is a timeliness-guaranteed process execution engine. The main structuring principles that were exploited to realize guaranteed timely service capabilities together with modularity and expandability are reviewed.

2.1 Basic components of process-structured real-time concurrent and distributed programs

The DREAM kernel as a *process execution engine* supports the following three types of concurrent and distributed program components:

- (1) Processes: In this paper, these are also called *application processes* (AP's) although in reality, some of these processes may play the roles of system management processes, e.g., processing managing I/O. The processes may frequently impose deadlines on the kernel for execution of their next computation-segments.

- (2) CREW (concurrent-read-&-exclusive-write) monitors: The CREW monitor is a shared data structure monitor and an extension of the *monitor* defined in [Bri77] in that the former possesses the *readers-writers* semantics (i.e., *concurrent-read-&-exclusive-write* semantics) instead of the *exclusive-read-&-exclusive-write* semantics associated with the latter.

- (3) Content-code (CC-) channels in the HU-DF scheme [Kim95a]: The *HU-DF (HU data field) scheme* for inter-process-group communication is an extension of the original *data field* scheme developed by Mori and other researchers in Hitachi, Ltd. [Mor93]. The essence of the data field scheme is to facilitate dynamic creation of multicast logical channels and dynamic connection of processes to the logical channels in such a way that the idiosyncrasies of the physical communication networks are transparent to the process designer. If the physical communication facility has the broadcast capability, then a logical multicast channel is facilitated by making all processing nodes using the channel to broadcast (through the physical communication facility) messages with the headers containing the ID of the channel called the content code (CC). The processing nodes connected to the logical channel can see all the messages coming through the physical broadcast facility but will pay attention only to those messages containing relevant content codes. The HU-DF scheme differs from the original data field scheme in that the former allows dynamic flexible connection of processes to the logical channels and supports not only conventional *event messages* but also *state messages* which are based on the distributed replicated memory semantics (a variation of the state message semantics in [Kop89]).

These three basic components represent fully general facilities for concurrent and distributed program structuring. Therefore, any operating system kernel that supports these three components along with various I/O

operations can be viewed as a general purpose kernel. Programs composed of the three types of components are called the PCD (process-CREW-DF) programs.

2.2 Five-layer structure and the principle of time-leasing machine layering

The architecture of the DREAM kernel is depicted in Figure 1.

As shown, the DREAM kernel adopts a unique approach for layering of its components. This special layering approach is the key to its realization of guaranteed timely service capabilities. The kernel consists of five layers in total and the bottom four of the five layers constitute the DREAM micro-kernel. Whereas the DREAM kernel can be viewed as a *process execution engine*, the DREAM micro-kernel can be viewed as a *kernel-thread execution engine*.

The kernel-thread, or thread for short, is an active concurrency unit operating inside the DREAM kernel. The set of threads is fixed at the operating system loading time. All the threads share the same address space. Except one called the *Main Thread (MT)* and responsible for selecting the next process to run and causing the process to run within the time periods allocated to itself (MT), all other threads are periodic threads (which behave like periodic time-triggered methods in RTO.k objects reviewed in Section 3.1). Once a thread is chosen to run, it can run for one time unit called thread time-slice. If a periodic thread chosen to run does not need the full thread time-slice, then it can “donate” the remaining portion of the time-slice to MT or just burn it by idling, depending upon the size of the remaining portion. Therefore, the thread scheduling is a simple low-overhead operation unlike the process scheduling. We believe that restricting all kernel threads except MT to be of this type only, i.e., periodic threads using one thread time-slice at a time, is a well justified approach for making the analysis of the worst-case response time of each thread to be simple without much sacrificing the hardware utilization.

The special layering depicted in Figure 1 is based on the organizational principle called the time-leasing machine layering, which is an important principle with respect to obtaining kernels with guaranteed timely service capabilities. Under this principle, the bottom layer, L0, owns the full power of the hardware machine. So, L0 uses the hardware machine at its own will. L0 contains a manager of a real-time alarm clock which supplies the current time upon receiving a request and also provides the “wake-up call” service. The remainder of the hardware machine time after L0’s use of the hardware machine, is “leased” to the next upper layer, L1. L1 then uses a portion of the hardware machine time it receives from L0 (i.e., the machine time which is not used by L0). Similarly, L2 uses a portion of the hardware machine time which is not used by either L0

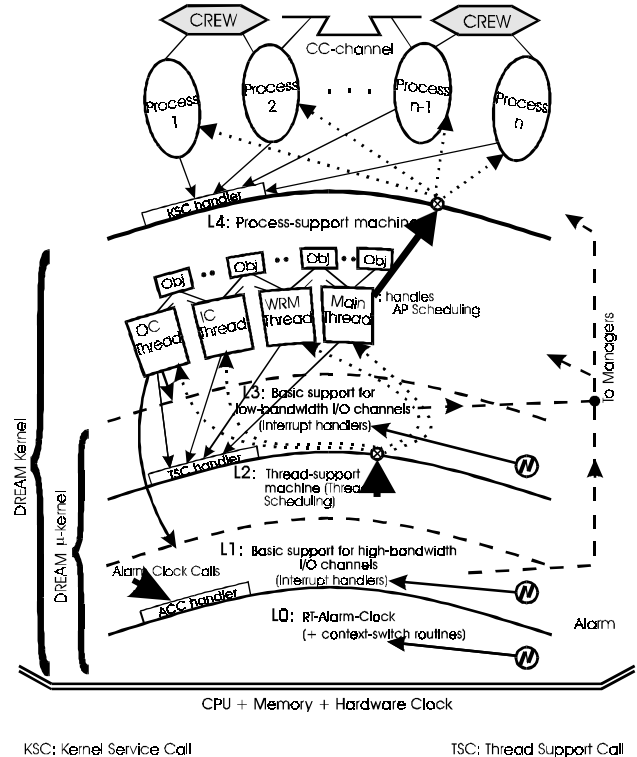


Figure 1. The architecture of the DREAM kernel

nor L1. This *recursive time leasing* relationship is applied to up to L4.

The upper four layers in the DREAM kernel contain the following components:

- (1) L1 contains basic support functions for high-bandwidth I/O such as LAN interface;
- (2) L2 contains the thread scheduler which is activated upon expiration of a thread time-slice as well as at some other times;
- (3) L3 contains basic support functions for low-bandwidth I/O such as serial character I/O;
- (4) L4 contains the process scheduler and other support functions for processes, CREW monitors, and CC-channels. The process scheduler is activated upon expiration of a *process time-slice* as well as at some other times. Here the size of the process time-slice is usually different from the size of the thread time-slice.

Another important part of the time-leasing machine layering principle is

to enforce that the amount of hardware machine time which is used by any layer during a basic response period be limited within a specific threshold.

The *basic response period* here refers to the maximum interval between the instant at which a process calls for a kernel service and the instant at which the service is completed. The obvious purpose of adopting this principle is to guarantee a certain amount of machine time being available to each of the upper layers.

The amount of machine time used by L0 is quite stable and the upper bound on L0's use during a basic response period can be relatively easily calculated. The upper bound on L2's use or L4's use during a basic response period can also be calculated without much difficulty. However, the cases of L1 and L3 which must respond to interrupts from external sources are different. In principle, the LAN interface manager in L1 can experience bursts of interrupts generated by the LAN interface due to an incoming message burst. We may discover that it becomes impossible to guarantee timely delivery of any non-trivial services by L4 to processes under such conditions of L1. If so, *the frequency of interrupt generations by L1 must be controlled and set not to exceed a certain threshold*. This means that once the frequency reaches the threshold, any further interrupts by the LAN interface which causes the interrupt frequency bound to be violated are disabled. This in turn means that some messages will simply be lost. That is a price paid for guaranteeing timely delivery of all non-trivial services to processes. We believe that this *dynamic control of interrupt sources*, which is a part of the time-leasing machine layering principle, is a requirement that is inevitable in any timeliness-guaranteed operating system.

2.3 Kernel-threads

As mentioned in the preceding subsection (2.2), kernel-threads, except MT, are periodic threads using one thread time-slice at a time. In the basic uniprocessor version of the DREAM kernel, there are four essential kernel-threads:

- (1) OCT (outgoing communication thread) which manages the sending of messages through the communication network,
- (2) ICT (incoming communication thread) which manages the distribution of messages coming through the communication network to the destination processes,
- (3) WRMT (watchdog-&-RTO-management thread) which manages the activation of object methods and checks if there are deadline violations, and
- (4) MT (main thread).

If the DREAM kernel were to operate as a process execution engine only, not as an execution engine for RTO.k objects, then only a part of WRMT, i.e., only the watchdog timer service, is needed.

Data input from a high-bandwidth device is realized through a combined effort of the device, the interrupt handler in L1, and the ICT in L4. There is usually a modest-size buffer, say B1, into which a high-bandwidth device can pour data until it is filled. The device generates an interrupt when the buffer B1 is filled. In response to the interrupt, the interrupt handler in L1 moves the data in B1 into another larger buffer B2 accessible to both the interrupt handler and the ICT. The interrupt handler may insert many data sets into B2 before the ICT becomes ready to access B2. Later when

the ICT gets the next thread time-slice, it moves the data in B2 to the destination, an AP.

One thing noteworthy here is that special application-specific threads can be introduced if fast response activities specific to the application must be supported. However, the set of all kernel threads must become fixed at the operating system loading time in order not to damage the ability to guarantee timely service capabilities of the kernel at a reasonable performance level.

2.4 Thread-to-thread atomic sections (TT-AS's)

Restricting kernel-threads, except MT, to be periodic threads using one thread time-slice at a time enables low-overhead scheduling of threads and easy analysis of such overhead. An additional measure taken to simplify the worst-case response time of a kernel thread is to restrict kernel-threads in the DREAM kernel to avoid conflicts among themselves in accessing shared data without using locks for the data. Instead, a thread needing to access a shared data structure orders the thread support machine (in L2) not to disturb the former, i.e., orders L2 to disable the thread switch so as not to let the machine power be taken away from the thread, until the thread notifies L2 that the disturbance prohibition period is over (and this obviously occurs when the thread finishes its access to the shared data structure). So, *even if a thread time-slice expiration occurs during the disturbance period (i.e., while the thread is accessing the shared data), the thread support machine does not change the running thread*. Such a code-segment during the execution of which a thread accesses shared data structures is called a thread-to-thread atomic section (TT-AS). The sending of a no-disturbance request to the thread support machine is called an "Enter-TT-AS" operation. Similarly, the cancellation of a no-disturbance request is called an "Exit-TT-AS" operation.

The TT-AS approach works only if the duration in which the thread executes the atomic section (AS) is much shorter than a thread time-slice. Generally this should be the case. Otherwise, either the thread time-slice was poorly chosen or threads and their shared data structures were not designed properly. This TT-AS approach is more efficient than the conventional data lock based approaches since kernel threads except MT are periodic threads and frequent changes of the running thread can occur under the latter approaches but not under the former approach.

2.5 Two-level scheduling

Since there are two types of concurrent computation-units, i.e., AP's and kernel threads, which the DREAM kernel deals with, scheduling of concurrent computation-units occurs in two different layers of the kernel. *The process time-slice should never be smaller than the thread time-slice*. Normally, the process time-slice should be multiple times as long as the thread time-slice is.

In L2 (the thread-support machine), the thread scheduler schedules the four threads, ICT, OCT, WRMT, and MT, for use of the hardware machine. Then in L4, the MT, when it is executing its core (i.e., AP Scheduler) (rather than an AP), schedules various AP's for use of the hardware machine. When an AP releases the control over the hardware machine voluntarily or under a mandate upon expiration of its process time-slice, the control goes to another thread (non-MT) or the core (AP scheduler) of the MT.

Also, in a manner analogous to the use of TT-AS's, AP's can use process-to-process atomic sections (PP-AS's). Therefore, synchronization between AP's can be realized via three different mechanisms: PP-AS, semaphore, and CREW monitor. Unlike in the case of kernel-threads, dynamic creation of AP's is also supported by the DREAM kernel.

Some advantages of this two-level scheduling approach are as follows. Scheduling of kernel-threads in a manner which is not interfered in any way by the AP scheduling makes it trivially easy to ensure that periodic kernel-threads provide essential services to the AP's precisely at the times determined at design-time. Also, the design of the MT-Core as a mediator among AP's considerably reduces the implementation complexity and also the context switching overhead to some extent over a direct AP-to-AP context switch design.

3. DREAM kernel as both a process execution engine and a real-time object execution engine

The remaining features of the DREAM kernel that do not belong to the process execution engine part, are the mechanisms directly related to supporting RTO.k objects. After a brief review of the essence of the RTO.k object structuring scheme in Section 3.1, those support mechanisms are discussed.

3.1 The RTO.k object structuring scheme

An initial abstract framework of the *RTO.k object model*, also called the time-triggered real-time object (TT RTO) model, came out of the attempt by the first co-author and Hermann Kopetz at the Technical University of Vienna to find a proper extension of the basic object model which is highly cost-effective in development of *hard-real-time* application systems. Based on the initial abstract framework formulated in late 1980's [Kop90], a concrete syntactic structure and execution semantics was developed in recent years [Kim94a, Kim94b, Kim94c].

The basic structure of an RTO.k object is depicted in Figure 2. It is an extension of the conventional object model(s) and two most important and unique extensions are the following:

(a) *Two clearly separated groups of methods:*

For some methods of an RTO.k object, a real-time clock serves as the mechanism for triggering the method executions as the clock reaches some values specified at

design time and such methods are called time-triggered (TT-) methods, also called the spontaneous methods (SpM's), and *clearly* separated from the conventional service methods (SvM's) triggered by messages from clients. The two types of methods in an RTO.k object are different not only in the way their executions are triggered but also in that

“actions to be taken at real times *which can be determined at the design time* can appear only in SpM's”.

Therefore, actions of the type “at constant-clock-value do S” or the type “sleep-until constant-clock-value” can appear only in SpM's. Incorporation of SpM's means introducing the potential for the following two new types of concurrent executions of object methods in addition to the potential for concurrent executions of SvM's that exist in conventional objects:

(Type I) Concurrency among SpM executions:

This concurrency is specified in an implicit but natural manner, e.g., two SpM's designed to be triggered at 10 am.

(Type II) Concurrency between SpM executions and SvM executions.

(b) *Basic concurrency constraint (BCC):*

In order to dramatically reduce the designer's efforts in guaranteeing timely service capabilities of RTO.k objects, the execution rule which prevents conflicts between SpM's and SvM's is incorporated. Basically, *activation of an SvM triggered by a message from an external client is allowed only when potentially*

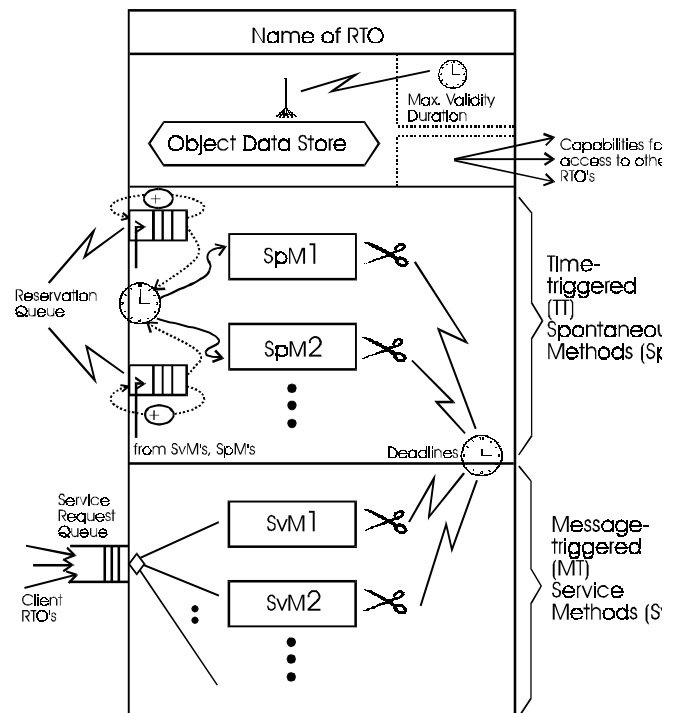


Figure 2. Structure of the RTO.k object model (Adapted from [Kim94c])

conflicting SpM executions are not in place. To be exact, when a message-triggered SvM is not free of conflict with an SpM in accessing the same portion of the object data space (ODS), execution of the former method (SvM) must not be allowed in a time zone earmarked for a TT-execution of the latter method (SpM). This restriction is called the basic concurrency constraint (BCC). Therefore, executions of SpM's are not disturbed by SvM executions and triggering times of SpM's are fixed at the design time. If a statement of the type "at 10am do S" appears in an SpM, its timely execution can be easily assured.

The above two features make the RTO.k object model clearly distinguished from other proposed real-time object models [Att91, Ish92, Tak92]. In addition, the RTO.k object contains the following features not found in the conventional object model(s):

(c) For each execution of a method of an RTO.k object, a deadline is imposed;

(d) Real-time data contained in an RTO.k object become invalid after the interval called the maximum validity duration passes.

Triggering times for SpM's must be fully specified as constants during the design time. Those real-time constants appear in the first clause of an SpM specification called the autonomous activation condition (AAC) section. An example of an AAC is

"for t = from 10am to 10:50am every 30min start-during (t, t+5min) finish-by t+10min" which has the same effect as

{ "start-during (10am, 10:05am) finish-by start_time+10min",

"start-during (10:30am, 10:35am) finish-by start_time+10min" }.

A provision is also made for making the AAC section of an SpM contain only candidate triggering times, not actual triggering times, so that a subset of the candidate triggering times indicated in the AAC section may be dynamically chosen for actual triggering. Such a dynamic selection occurs when an SvM within the same RTO.k object requests future executions of a specific SpM.

An underlying design philosophy of the RTO.k object model is that an RTCS will always take the form of a network of RTO.k objects. The designer of each RTO.k object provides a guarantee of timely service capabilities of the object by indicating the deadline for every output produced by each SvM (and each SpM which may be executed on requests from SvM's) in the specification of the SvM (and some relevant SpM's) advertised to the designers of potential client objects. Before determining the deadline specification, the server object designer must convince himself/herself that with the object execution engine (hardware plus operating system) available, the server object can be implemented to always execute the SvM such that the output action is

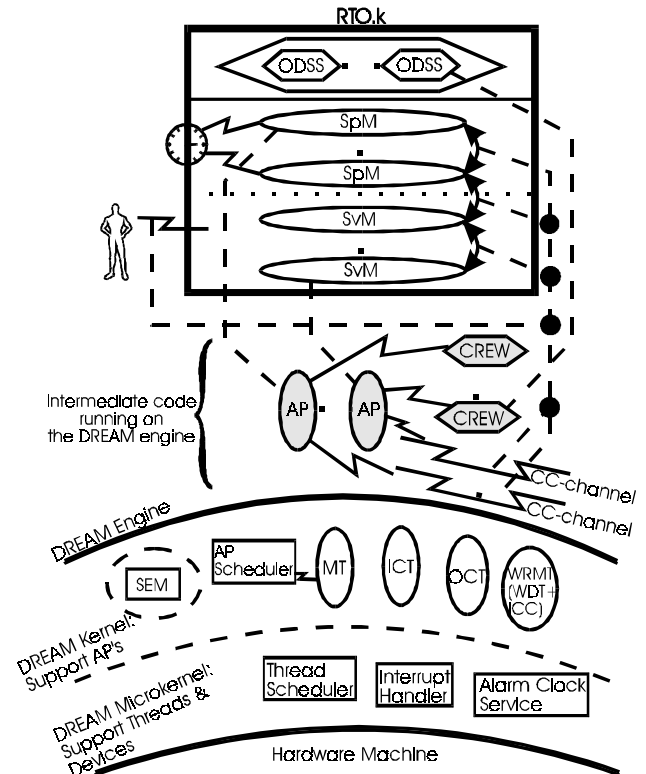


Figure 3. Mapping an RTO.k object to a PCD-program

performed within the deadline. Again, the BCC contributes to major reduction of these burdens imposed on the designer.

The RTO.k object model is effective not only in the multiple-level abstraction of real-time (computer) control systems under design but also in the accurate representation and simulation of the application environments. In fact, it enables uniform structuring of control computer systems and application environment simulators [Kim94b] and this presents considerable potential benefits to the system engineers.

3.2 A mapping of an RTO.k object to a PCD-program

When the DREAM kernel executes RTO.k objects, it actually executes equivalent PCD-programs, i.e., programs composed of processes, CREW monitors, and data fields (i.e., CC-channels). Figure 3 depicts the mapping relationship between components of an RTO.k object and the corresponding components of an equivalent PCD-program. As shown,

- (1) object methods, both SpM's and SvM's, are mapped to processes,
- (2) ODS segments (ODSS's) to CREW monitors,
- (3) access paths to SvM's to CC-channels, and
- (4) result-return paths to clients to CC-channels.

This way of utilizing CC-channels facilitates the transparency of object locations.

When an SpM is mapped to a process, the process must present to the DREAM kernel (1) the AAC of the

SpM, (2) the associated deadline specification, and (3) access rights of the SpM for ODSS's. Similarly, when an SvM is mapped to a process, the process must present to the DREAM kernel (1) the associated deadline specification, (2) access rights of the SvM for ODSS's, and (3) other information related to its pipelined execution possibilities.

As mentioned earlier, this execution approach has the advantages of utilizing a general-purpose kernel (such as the DREAM kernel) supporting various types of concurrent and distributed application software.

3.3 The watchdog and real-time object management thread (WRMT) supporting RTO.k objects

Of the kernel-threads shown earlier in Figure 1, the WRMT provides services needed mainly to support RTO.k objects. It is a periodic kernel-thread responsible for:

- (1) timely activation and future reservation of SpM's,
- (2) activation of SvM's upon receiving of the corresponding service requests,
- (3) enforcing the basic concurrency constraint, and
- (4) deadline checking of the methods and invoking of an appropriate exception handler upon detection of a deadline violation.

A queue in layer L4, called the Reservation Queue (RvQ), holds the activation schedules of SpM's during next t time units. Reservations of SpM's are done by inserting the earliest start time and the latest start time of each SpM into the RvQ, which is to be activated during the next time period, p . The WRMT activates SpM's from the RvQ at their scheduled times by inserting the SpM's into the Ready AP Queue (RAPQ) and makes future reservations into the RvQ. The WRMT also activates an SvM upon receiving a service request message by inserting the SvM into the Ready AP Queue only if there is no possibility for the SvM to run into an ODS-conflict with any SpM. If an SpM which can run into an ODS-conflict with a SvM requested by an external client object is to be activated in d time units where d is less than the worst-case execution time of the SvM, the WRMT will not activate the execution of the SvM.

Use of the WRMT for supporting RTO.k object methods is an approach striking a good balance between the response time and the overhead, especially in view of the possibilities of using a dedicated process for similar functions or using the real-time alarm clock in L0 for some parts of the functions (e.g., TT-activation of an SpM).

3.4 Extensibility: incorporation of additional kernel-threads and system processes

As new I/O and networking devices are incorporated, the DREAM kernel can be expanded in several ways. First, device interrupt handlers must be inserted into layer L1 or L3 depending upon the data

transfer bandwidth of the new device and the required response to a request of the device for an attention. Then, a decision must be made whether to impose additional responsibility for supporting the new device on the ICT or to introduce a new dedicated kernel-thread or to introduce a new dedicated AP.

4. A prototype implementation of the DREAM kernel and implementation techniques adopted

The first prototype DREAM kernel, v.D2, was implemented by the co-authors to run on a network of PC's connected via Ethernet and equipped with Intel 80486 processors, DOS-BIOS device drivers, and the Packet Ethernet driver. Services of the DREAM kernel including process management services can be requested from within a C++ program (representing an implementation of an RTO.k object) via calls for DREAM library routines.

4.1 Internal RTO.k object structuring

All components of the DREAM kernel v.D2 were structured in OO-forms, many times in RTO.k object forms. Initially, the fact that the naturalness and the representational power of the RTO.k object model were prominently shown even in this context looked very interesting. However, through subsequent reflections, we have come to the realization that a kernel is nothing but another kind of a real-time program and thus the applicability of the RTO.k object structuring scheme to the implementation of the DREAM kernel v.D2 should have been expected if the RTO.k scheme were indeed an appropriate approach for structuring NG RTCS's.

4.2 Inter-process-group communication facilities

In the HU-DF scheme [Kim95a], a process can freely connect itself to or disconnect itself from a multicast logical channel, which is somewhat restricted in the original DF scheme.

The DREAM kernel not only supports interprocess communication through CC-channels but also supports communication through ports [Ras89]. A port is a message communication channel with only a single AP as a receiver. An AP may notify its intention to the kernel to function as a receiver to a specific port. In such a case, the kernel makes sure that no other AP's are granted the receive rights to that port.

4.3 Layer L0 services

Layer L0 provides the following essential services to the upper layers :

- (1) time-slice generation,
- (2) alarm clock service to upper layers,
- (3) context switch. It is essential to include the context switch routine in L0 because during a context switch, a timer interrupt service routine must not be invoked. In fact, the context switch routine is the only routine in the

entire DREAM kernel v.D2 that is written in an assembly language.

4.4 An RTO.k structured application running on DREAM kernel v.D2

A preliminary version of the prototype DREAM kernel was used in an experimental development of a defense system with its application environment simulator [Kim94b] and has since been replaced by the current version (v.D2). This real-time distributed application software consists of nine different types of tailorable RTO.k objects and runs on a network of five or more PC's. In this experimental effort, translation of the RTO.k object structured program into an equivalent PCD-program was done manually. Some real-time fault tolerance capabilities were also implemented. The application software consists of approximately 30,000 lines of C++ code. This experiment gave us a considerable amount of confidence in the appropriateness of the architecture of the DREAM kernel as well as the power of the RTO.k object structuring scheme.

5. Conclusion

At present, the DREAM kernel is little more than a minimal-functionality model of a timeliness-guaranteed operating system kernel supporting real-time processes and RTO.k objects. It can be extended in several major directions. The most obvious among them is to extend it to utilize multi-processor architectures and highly parallel machine architectures. Adapting the DREAM kernel to commercial micro-kernel environments is another meaningful direction to pursue. We plan to implement in the near future another version of the DREAM kernel which uses the IBM micro-kernel as its component. Realization of NG real-time computing will require investment of sizable efforts by industry to revise and extend their existing operating system products to possess guaranteed timely service capabilities and it is our hope that industry find some of the approaches and principles exploited in the DREAM kernel useful in such efforts.

Acknowledgment: The research work reported here was supported in part by US Navy, NSWC Dahlgren Division under Contract No. N60921-92-C-0204, in part by the California Transportation Department via the UCI Institute for Transportation Studies, in part by the University of California MICRO Program under Grant No. 93-080, in part by Hitachi, Ltd, in part by Postech, and in part by ETRI. The efforts of L. F. Bacellar were also supported by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - CAPES and by Universidade Federal do Rio de Janeiro - UFRJ both from Brazil.

References

[Att91] Attoui, A. and Schneider, M., "An Object Oriented Model for Parallel and Reactive Systems",

Proc. IEEE CS 12th Real-Time Systems Symp., 1991, pp. 84-93.

[Bri77] Brinch Hansen, P., "The Architecture of Concurrent Programs," Prentice-Hall, 1977.

[Ish92] Ishikawa, Y., Tokuda, H., and Mercer, C. W., "An Object-Oriented Real-Time Programming Language", *IEEE Computer*, October 1992, pp. 66-73.

[Kim94a] Kim, K.H. et al., "Distinguishing Features and Potential Roles of the RTO.k Object Model", *Proc. 1994 IEEE CS Workshop on Object-oriented Real-time Dependable Systems (WORDS)*, Oct. '94, Dana Point, pp.36-45.

[Kim94b] Kim, K.H. and Kopetz, H., "A Real-Time Object Model RTO.k and an Experimental Investigation of Its Potentials", *Proc. 1994 IEEE CS Computer Software and Applications Conf. (COMPSAC)*, Nov. 1994, Taipei, pp.392-402.

[Kim94c] Kim, K.H., "A Utopian View of Future Object-Oriented Real-Time Dependable Computer Systems", (Invited paper) *Proc. 1st Int'l Workshop on Real-Time Computing Systems and Applications*, Seoul, Korea, Dec. 1994, pp. 59-69.

[Kim95a] Kim, K.H., Mori, K., and Nakanishi, H., "Realization of Autonomous Decentralized Computing with the RTO.k Object Structuring Scheme and the HUDF Inter-Process-Group Communication Scheme", *Proc. 1995 IEEE CS Int'l Symp. on Autonomous Decentralized Systems (ISADS)*, April 1995, Phoenix, pp.305-312.

[Kim95b] Kim, K.H., "Toward New-Generation Real-Time Object-Oriented Computing", *Proc. IEEE CS 5th Workshop on Future Trends of Distributed Computing Systems (FTDCS)*, Cheju Island, Aug. '95, pp.520-529.

[Kop89] Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Wolfgang, S., Senft, C., and Zainlinger, R., "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach", *IEEE Micro*, Feb. 1989, pp. 25-39.

[Kop90] Kopetz, H. and Kim, K.H., "Temporal Uncertainties in Interactions among Real-Time Objects", *Proc. IEEE Computer Society's 9th Symp. on Reliable Distributed Systems*, Huntsville, AL, Oct. 1990, pp.165-174.

[Mor93] Mori, K., "Autonomous Decentralized Systems: Concept, Data Field Architecture, and Future Trends", *Proc. IEEE CS Int'l Symp. on Autonomous Decentralized Systems (ISADS 93)*, Mar. 1993, Kawasaki, Japan, pp. 28-34.

[Ras89] Richard Rashid, Robert Baron, Alessandro Forin, David Golub, Michael Jones, Daniel Julin, Douglas Orr, Richard Sanzi, "Mach: A Foundation for Open Systems", *Proc. the IEEE Second Workshop on Workstation Operating Systems (WWOS2)*, September 1989, pp.109-113.

[Tak92] Takashio, K., and Tokoro, M., "DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems", *Proc. OOPSLA*, 1992, pp. 276-294.