Global Journal of Computer Science and Technology
Volume 12 Issue 2 Version 1.0 January 2012
Type: Double Blind Peer Reviewed International Research Journal
Publisher: Global Journals Inc. (USA)
Online ISSN: 0975-4172 & Print ISSN: 0975-4350

Architectural principles support the continuous multimedia applications

By Dr.L.V.Reddy, J.Durga
Department of. CSE, SVEC,A.Rangampeta

Abstract - In this paper we discuss some of the architectural principles which are useful to support the continuous media applications in a microkernel environment. In particular, we discuss i) the principle of upcall-driven application structuring whereby communications events are system rather than application initiated, ii) the principle of split-level system structuring whereby, key system .functions are carried out co-operatively between kernel and user level components and iii) the principle of decoupling of control transfer and data transfer. Under these general headings a number of particular mechanisms and techniques are discussed.

GJCST Classification: H.5.1

Architectural principles support the continuous multimedia applications

Dr.L.V.Reddy [α], J.Durga [Ω]

Abstract - In this paper we discuss some of the architectural principles which are useful to support the continuous media applications in a microkernel environment. In particular, we discuss i) the principle of upcall-driven application structuring whereby communications events are system rather than application initiated, ii) the principle of split-level system structuring whereby, key system .functions are carried out co-operatively between kernel and user level components and iii) the principle of decoupling of control transfer and data transfer. Under these general headings a number of particular mechanisms and techniques are discussed.

I. Introduction

We are interested in both communications and processing support for distributed real-lime/multimedia applications in end systems, and believe that such applications require thread-to-thread realtime support according to user supplied quality of service (QoS) parameters. Such support, depending on the level of QoS commitment required, may require dedicated, per-connection, resource allocation the CPU scheduler, virtual memory system and communication system. It may also require ongoing dynamic QoS management in all these areas[1]. Another important requirement we have imposed on ourselves is to support standard UNIX applications on the same machine as our real-time/ multimedia support infrastructure; we do not want to build a specialist real-time system that is isolated from the standard application environment. In our paper the prime consideration is the efficiency. In particular in minimizing system imposed overheads by reducing the cost and number of system calls, context switches and copy operation.

In this paper is structured as three main sections, each of which describes a key architectural principle of our design. The three principles are: i) up call driven application structuring whereby communications events are system rather than application initiated, ii) split-level system structuring whereby key system functions are carried out cooperatively between kernel and user level components and iii) decoupling of control transfer and data transfer whereby the transfer of control is carried out asynchronously with respect to the transfer of data.

Author [α] : Professor, Department of CSE, SVEC,A.Rangampeta.
E-mail : lakkireddy.v@gmail.com
Author [Ω] : Asst.professor, Department of. CSE, SVEC,A.Rangampeta.
E-mail : durgapriyadharshini@gmail.com

II. Upcall-Driven Application Structuring

In conventional designs, system APIs are mostly passive and applications are mostly active. For example, when an application needs to send or receive data, it typically invokes a system call such send() or recv(). It also provides the buffer from/to which data is to be sent/received. In contrast, our continuous media API is structured so that the system infrastructure is active and applications are passive. Application programmers attach rthandlers, which are C functions containing application code to process the real-time media, to rtports, which are globally unique units of addressing. Then, programmers establish connections with a given QoS between rtports. At connect time, the system, rather than the application, allocates buffers for connections ,and provides the thread on which the rthandlers will be executed. At data transfer time, the system decides to upcall the application to obtain/ deliver data at instants determined by the QoS specification (in terms of rate, jitter, delay etc.) provided by the application at connect time. When an application rthandler is upcalled, the address of the associated rtport's buffer is passed as an argument so that application code in the rthandler can access the buffer. Source rthandlers are expected to fill buffers with data to be sent, and sink rthandlers to use the data as provided.

In conventional designs, systems API are mostly passive and applications are mostly active. For example, when an applications needs to send or receive data, it typically invokes a system call such as send() or recv(). It also provides the buffer from/ to which data is to be sent/received. In contrast, the continuous media API is structured so that the system infrastructure is active and applications are passive.

There are three major benefits of style of an application/system interaction in our context. First, it relieves the application of the burden of explicitly creating threads and allocating buffers. Second, the system, rather than the application, can choose the timing of application code execution, and thus can optimally monitor and manage the Quality of Service (QOS) of the connection, including the execution of application code, to provide the required thread-to-thread QOS support. Third, the structuring of the API with the API rthandlers is a natural and effective model for real-time programming. Real-time programming is considerably simplified when programmers can structure applications to react to events and delegate to the system the responsibility for initiating communication events. The programmer is still ultimately in control of event initiation but this control is expressed declaratively through the provision of QOS parameters at connect time and need not be explicitly programmed in a procedural style[2].

Along with these benefits, an efficiency gain potentially results from upcall-driven application structuring, because a single thread ca be used for both protocol and application processing. In conventional systems, application interface with communications by performing system calls which block and reschedule if the communications system is not ready to send or if data has not yet arrived. With infrastructure initiated communication, on the other hand, it is not necessary for the application and communications system to wait for each other, and thus no context switch is incurred, as the communication system always initiates the exchange and the application code is always be ready to run.

III. Split-Level System Structuring

The distributed multimedia applications will require high degree of internal concurrency. For example, it is likely that each media stream will require at least one thread of execution and it is also likely that applications will be structured as pipelines of processing stages on streams of media. Split-level structuring is used to maintain the merits of user space management while mitigating its demerits.

IV. Split Level Scheduling

The above merits and demerits are particularly evident in the case of CPU resource management through user level threads. Here the benefits are cheap user level concurrency and the drawback is that the relative urgencies of threads in different address spaces are not visible to the kernel scheduler.

In split level scheduling, a small number of virtual processors (VPs) execute user threads in each address space. The split level scheduling schemes maintains the invariant that [5]:

- Each user level scheduler (ULS) always runs its most urgent user thread, and
- The kernel level scheduler (KLS) always runs the VP supporting the globally most urgent user thread.

Split level scheduling allows many contexts switches to take place cheaply in the same address space but also ensures that the relative urgencies of threads across the whole machine are appropriately taken into account.

V. Split Level Communication

The strategy of split level communication structuring is a leave the kernel responsible for multiplexing and demultiplexing network packets to application address spaces, but the application address spaces perform transport level processing. In this way, transport protocol processing can automatically take advantages of the split level scheduling infrastructure and thus exploit cheap user level context switches.

Split level communication structuring also allows meaningful deadlines to be placed on (transport level) protocol processing activities, as the ultimate deadline of the final packet delivery is easily available in the application context. Thus, the scheduling of protocol processing need not be performed 'blind' as it is in typical kernel implementation. Another advantage is that multiple transport protocols can easily be dynamically configured in and out of applications according to their particular requirements. This is important in a multimedia context where different protocols may be appropriate for different media types.

VI. Split Level Buffer Management

The strategy of split level buffer management is for the kernel level manager to 'loan' physical, locked, buffers to per-address space managers, but to reserve the rights to reclaim the buffers if memory is urgently required or to retain the buffer longer than it has agreed to. The policy adopted in the design level is that the application is allowed to keep buffer for at least the normal duration of transport protocol processing time plus rthandlers execution time. If the period has elapsed and the application space has not returned ownership of the buffer to the kernel, the kernel may reclaim the buffer.

The semantic of 'reclaiming' locked buffers is to convert locked memory into standard swappable virtual memory. In this way, applications do not lose their data although they do lose guaranteed access latency to that data as the memory region is subject to being paged out. If the kernel does not need to reclaim buffers at the end of an rthandler execution, the user space manager may reuse buffers for other connections.

VII. Decoupling of Control Transfer and Data Transfer

In traditional systems, the transfer of control and the transfer of data are usually tightly coupled. For example, the execution of a UNIX system calls passes data to the kernel and simultaneously transfers control to the kernel.

VIII. Direct Connections

In distributed multimedia applications it is often required to receive continuous media data from the network and directly play it out on a device such as an audio card or a frame buffer (which is probably

Global Journal of Computer Science and Technology Volume XII Issue II Version I January 2012

managed by kernel level code). The application may or may not require keeping track of the transfer of individual buffers of data for synchronization purposes. The opposite scenario, where data from a local device is to be put directly onto the network, is equally common. In conventional operating system, the only way to achieve such a data flow is to route the data through an intermediate user process. But it involves significant per-buffer overheads [2].

In a direct connection, data that is to be passed directly between the network and a local device does not pass into user space at all; it is processed entirely within kernel space. When a direction connection is established, the infrastructure pre-maps the buffer associated with the connection into the output device's memory. Then, the data can be directly copied off the network card on to the device without leaving kernel space. The fragmentation/re-assembly functions of the in-kernel transport protocol.

If the user application does not need to synchronize with the delivery of buffers, no further overhead is incurred. However, if it is required to synchronize, the application can attach an rthandler to the rtport. This is up called on each buffer transfer with the usual rthandler semantic. The only difference in the API between this case and the normal case is that buffer pointer passed as an argument to the rthandler upcall will be a null pointer as the application context will not have the rights to directly access the kernel managed buffer.

IX. Asynchronous System Calls

For continuous media connections asynchronous system calls exploit the predictable periodicity of the transfer of control and data between application address spaces and the kernel. To issue an asynchronous system call (e.g. an asynchronous version of send()), user level library code:

- Places an operation identifier and parameters in the shared KLS/ULS memory bulletin board area and then
- Sets an 'operation request' bit, also in the bulletin board area.

The KLS, when it runs at the next system clock tick, notices that an operation bit is consequently passes the user's parameters to a kernel server thread which carries out the system call on behalf of the ULS. This avoids a special domain crossing for the system call at the expense a bitmap on each clock interrupt.

X. Asynchronous Software Interrupts

The implementation of software interrupts is similar to that of asynchronous system calls and similarly avoids a special domain crossing. The mechanism for kernel to VP control transfer is as follows:

- The KLS places an event identifier and parameters in the KLS/ULS bulletin board area
- The KLS alters the program counter field of the target VP's context structure points to a standard entry point in the ULS.

Thus, when the VP is next scheduled, the VP immediately enters its ULS, which picks up the event identifier and parameters, and schedules a user thread to deal with event. Asynchronous software interrupts are also provided as a service accessible from user level code. This service enables library code in one address space to cheaply notify an event to another address space on the same machine. The service also allows the sender to name a pre-existing memory segment shared between the sender and receiver address spaces so that data can be optionally transferred in the same call.

XI. User Level Pipelines

The API for pipelines of processing stages is very similar to the connection abstraction. But rather than passing a pair of rtports as arguments to the connect() primitives, we pass a list of rtports. In the case of pipelines, the delay QOS parameter applies end-to-end over the entire chain of rtports.

Intermediate processing stages in pipelines are also realized in a similar way to that described above: When data arrives at an intermediate processing stage, the rthandler returns, it is assumed that the rthandler's application code has performed some appropriate processing on the buffer whose address was passed up to it, and the data can be passed on to the next page.

As the various stages of a pipeline form part of the same application, it is typically the case that pipelines are implemented in a single address space. The data transfer mechanism in this case is as follows: When an rthandler implementing one stage of a pipeline returns, having operated on a buffer, the next stage in the pipeline is simply passed the address of the same buffer. Meanwhile, the first stage sets of work on a second buffer; and so on. At the end of the pipeline, when buffers are finished with, they are returned to a user level pool from which that can be reused by the first pipeline stage. With this implementation, intra address space pipelines incur only user level control transfers between the threads dedicated to each pipeline stage, and zero copy operations between stages. The API is the identical regardless of whether intra-address space, inter-address space or inter-machine connections or pipelines are used. Inter-address space communication on the same machine uses buffers that are statically mapped into both the source and sink address spaces for data transfer, and use asynchronous software interrupts.

XII. Proposed Scenario

The integrated use of the principles and techniques described above, the scenario is illustrated in figure 1, involves the transfer of compressed video from a frame grabber card on a source machine to a decompress/display application on a sink machine. In figure 1, the large ovals represent user address spaces with library code below the horizontal line and application code above. The rectangles represent kernel space with the enclosed shaded regions representing devices.The send side features a direct connection, involves the video capture device and the network interface, which avoids the need for data to pass into user space. It also features the (optional) use of an rthandler to allow the sender, which is structured as an upcall-driven application, to monitor and synchronize with the progress of the connection.

On the receive side, the split level buffer management system allocates a physical buffer from the kernel buffer pool to hold incoming network packets associated with the connection. This buffer is statically mapped into both kernel space and the application address space.In the split level communication system, when a complete network level packet has been received, the application address space's ULS is notified via the conditional deadline mechanism and initiates transport level processing.

This may involve the receipt of further network packets to build a complete user level buffer.When a complete user buffer has been built, and the receiving thread has the globally earliest deadline, the ULS runs a thread which upcalls the application's rthandler with the address of the buffer as a parameter. The receive side features a user level pipeline which involves one user thread performing decompressed and another displaying uncompressed video in a window. The display is achieved by means of asynchronous system calls to display device. Context switches between the two pipeline threads are achieved at user level costs and the transmission of data from the decompressor to the displayer does not involve data copying.

XIII. Conclusions

We discussed three architectural principles useful for the support of distributed real-time multimedia application in operating system. Firstly we contended that the principles of upcall-driven application structuring leads to well structural real-time applications, relieves applications of the burden of explicit thread creation and buffer allocation, and leads to potential efficiency gains because of reduced context switches.

Secondly we argued for the principles of split-level system structuring. We suggested that it can improve efficiency by exploiting application specific knowledge (e.g. scheduling deadline or buffer requirements) in a local, user level, context where application/manager interact6ion is cheap, while relying on a kernel level manager to 'bias' resources to application address spaces of the basis of their relative needs. Active co-operation of management information between user and kernel level managers is key, but as long as an asynchronous style of communication between managers is acceptable, this can be achieved cheaply by means of a shared memory 'bulletin board'.

Finally we discussed the three principles working together in this paper. They are capable of exploited in a range of operating system environments. Similarly many of individual techniques can be useful in a stand alone fashion. When it can be compared with the distributed level, the principles which we have discussed validated in terms of direct performance measurements.

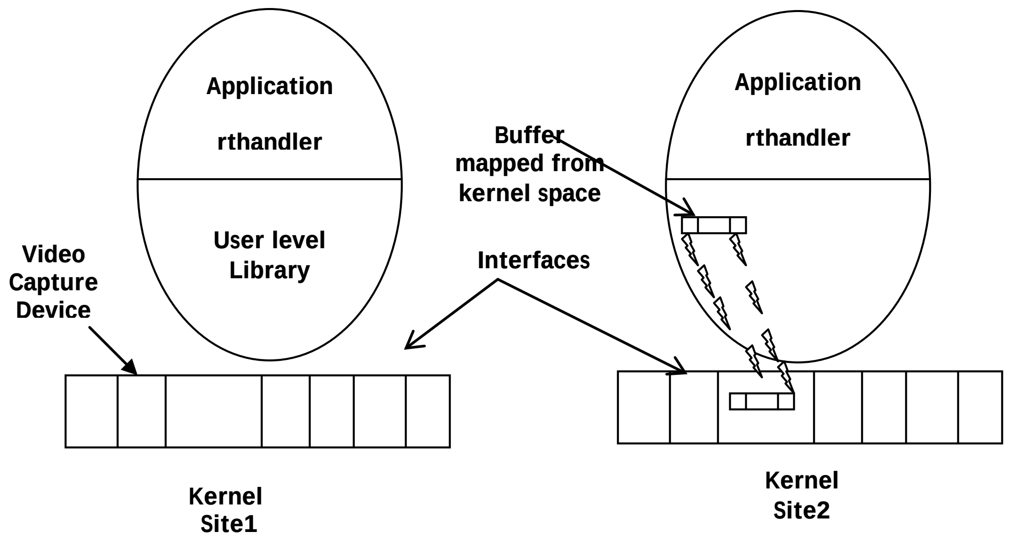

Figure 1 : An illustrative Scenario

References References Referencias

1. A.P. Black, 1983, 'An Asymmetric Stream Communication System', Proceedings of 9th ACM Symposium in Operating System Principles', pp 4-10.
2. M. Acceta, A. Tevanian and M. Young, 1986, 'Mach: A new Kernel Foundatiuon for UNIX Development', Technical Report, Camegie Mellon University.
3. Geoff Coulson and Gordon Blair, 1995, 'Architectural Principles and Techniques for Distributed Multimedia Application Support in Operating Systems', ACM SIGOPS Operating Systems Review, Vol.29 (4), pp17-24.
4. Geoff Coulson and Gordon Blair, 1994, 'Micro-kernel Support for Continuous Media in distributed Systems', Computer Networks and ISDN Systems, Vol 26 (10), pp 1323-1341.
5. Geoff Coulson, Gordon Blair, P. Robin and D. Shepherd, 1994,'Supporting Continuous Media Applications in a Micro-Kernel Environment', Proceedings of the 1st International Workshop on Architecture and Protocols for High-Speed Networks, pp 215-234.
6. Ramesh Govindan, D.P. Anderson, 1991, 'Scheduling and IPC mechanisms for continuous media', ACM SIGOPS Operating Systems Review, Vol. 25 (5), pp 68-80.
7. C.L. Liu and J.W. Layland, 1973, 'Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment', Journal of the ACM, Vol. 20(1), pp 46-61.
8. C.A. Thekkath, T.D. Nguyen, 1993, 'Implementing network protocols at user level', Proceedings on Communications architectures, protocols and applications, pp 64 -73.

This page is intentionally left blank