Admission Control, Reservation, and Reflection in Operating Systems[1]

Prof. John A. Stankovic
Department of Computer Science
University of Virginia
Charlottesville, Virginia

# 1 Introduction

Current trends in operating systems include adding real-time features for support of audio and video [3], exposing lower level details to the application level [2], and pushing functionality and application level information into the kernel [1, 4]. It is possible to consider all of these things and other related issues as being synergistically combined via the concepts of admission control, resource reservation, and reflection. I propose that these three concepts become a central principle in the design of operating systems. In many systems that have direct support for multimedia, admission control and resource reservation, these are already central principles. The need for this view is becoming greater and greater as more real-time features are being supported by general purpose OSs. I will discuss each of these concepts, highlight their interactions, provide motivation for their value as a central principle, and show that these principles already exist in the Spring real-time kernel [8, 9].

# 2 Admission Control

Admission control is a function that decides if new work entering the system should be admitted or not. The key ingredients of admission control include a model of the state of system resources, knowledge about the incoming request, the exact algorithm to make the admission control decision, and policies for the actions to take upon admission and upon rejection.

Probably the earliest form of admission control in OSs is the adaptive scheme of trying to maintain a certain degree of multiprogramming to avoid

[1]Appeared in *IEEE Bulletin of the Technical Committee on Operating Systems and Application Environments (TCOS)*, Vol. 10, No. 2, Summer 1998.

thrashing and associated poor performance. In this case, the model of the state of the system is based on the busyness of the system and is usually CPU utilization. No knowledge about the incoming request is usually known. The algorithm is simply that if the system gets too busy, then work is prevented from being admitted *for now*. Upon admission the work is placed on a ready queue and if it is not admitted now, it waits until the system utilization subsides below some threshold. Since there are no time constraints involved all work is eventually admitted. For purposes of comparison to the next case below, note that no information about the new work is known (i.e., there is no reflective information) and no reservations are made when work is admitted. No lower level details are exposed to the application, and no information is pushed to the application.

Now consider hard real-time systems. Many hard real-time systems are statically scheduled and operate in highly deterministic environments. This facilitates the timing analysis required of these systems. But, many hard real-time systems operate in dynamic environments where static scheduling is too costly. What is required is a solution that enables on-line careful timing analysis and dynamic scheduling. A solution provided in the Spring kernel [9, 10] included the synergistic combination of admission control, resource reservation, and reflection; so this concept already exists in the hard real-time domain. Here the model of the state of the system is a detailed time line that identifies the start time and finish time (based on a worst case execution time model) for each admitted task on each resource that it requires. Significant reflective information is known about each incoming task because they are pre-analyzed for a particular real-time system; there are no general purpose on-the-fly tasks created. The reflective information known about the requested work includes worst case execution time, shared data required by this task, precedence constraints, importance level, which tasks communicate with this task, deadline, etc. The algorithm is a heuristic that schedules the task on the detailed time line along with all the previously admitted tasks in such a manner that if successful all the tasks will meet their deadlines. See [6, 5] for a detailed description of the algorithm. If the task is admitted it has been assigned a very specific time slice (although it may actually execute early under certain conditions [7]). If it is not admitted then a separate policy is invoked to decide what action to take. Typical actions include: try a simpler version of the task if any exists, or if the deadline is far away try to schedule the task on another node, or if the deadline is close then just reject this task. These policies can be modified based on the importance of the task. The low level details of the entire guaranteed schedule are available to the application. A large amount of application semantic information is pushed into the kernel (via the compiler and a special system description language). For example, the Process Control Block (PCB) contains, in addition to the typical information, worst case execution times, deadline information, precedence requirements, a communication graph, fault tolerance information, etc.

More recently, work on supporting quality of service for audio and video has also used admission control and reservations. In many systems various amounts and types of reflective information are also used, but it is rarely identified as a central principle. The typical model of the system has been utilizations identified independently for multiple resources such as CPU, network bandwidth, disk, and memory. The precise admission control algorithm has varied from system to system, but it is usually based on some simple sum of utilizations of all previously admitted work together with the requirements of the new work. If the utilizations are less then 100% then the work is admitted, else some renegotiation might take place and the work admitted with less guaranteed service. Application level information regarding peak loads, end-to-end delays, jitter requirements, etc. are pushed down into the network layers. Performance data may be pushed up to the application level if renegotiation is required.

# 3 Resource Reservation

Resource reservation is the act of actually assigning portions of resources to a task. In the early OS work for maintaining a reasonable level of multiprogramming, *no resource reservation* was done.

In the Spring kernel explicit resource reservation was done very precisely on each type of resource and in a coordinated manner. For example, if TASK A has the CPU reserved from time 10 to 15 this task would also be assured to have exclusive access to any needed resources including shared data structures during that same specific time interval. In fact, this integrated resource reservation is orchestrated so carefully that semaphores are not required (i.e., resource conflicts are avoided via scheduling). Such precise resource reservation is valuable for a hard real-time system where careful timing guarantees are required. However, this scheme is too expensive to be used for audio and video processing.

To support audio and video processing percentages of resources are reserved independently. Some resources such as memory might be explicitly reserved, while other resources such as cpu or network bandwidth might be reserved in the form of percentages. In some sense, resource reservations need to have application information (what data structures does a task need, or how much cpu does an application require) pushed down into the OS so that application specific performance can be attained. For renegotiation of QoS, OS level performance data is pushed up to the application so that it may modify its requirements if possible (and obtain new reservations).

# 4 Reflection

Reflective information is meta data about the system. It can be meta data about the application and it can be meta data about the performance and properties of the OS (or microkernels). All OS are reflective to some degree. However, if we elevate the notion of reflection to a central principle of the system then it is possible to build more flexible systems [9, 10, 11], and highlight the need to choose the right reflective information for a given system. Some of this flexibility can be used to obtain better performance, e.g., better performance to meet real-time constraints.

In the degree of multiprogramming example used above, there is no reflective information.

In the Spring kernel there is significant reflective information which enables on-the-fly performance guarantees in a dynamic setting. Most of the reflective information is from the application down to the OS. Some work was also undertaken to make use of reflective information from the kernel (regarding schedules) back to the application (agile manufacturing), but this is beyond the scope of this paper.

The example of supporting QoS for audio and video also makes use of reflective information about the needs of the various audio and video streams. This is usually in the form of bandwidth, peak load, end-to-end, and jitter requirements. In addition, there is no reason why the OS itself can't make reflective information about the OS available to applications as is used in systems like the exokernel work [2] and in various thread packages [1, 4]. Because all this information is available to admission control and renegotiation policies, better application specific performance is possible and reservations can be dynamically adjusted if necessary.

# 5 Summary

Many of today's applications are becoming so demanding that knowledge about them (including timing constraints) seems to be necessary to obtain good performance. This seems to have always been the case in real-time systems, and now, if effect, almost all systems are becoming real-time systems. The framework of admission control, resource reservation, and reflection is able to highlight the necessary ingredients to obtaining good performance. Focusing on this framework will encourage designers to identify the proper reflective information that needs to be pushed between the application and the kernel and vice versa, and to choose appropriate admission control (and renegotiation), and reservation schemes. It is likely that the answers will be different based on the type of system being built (hard real-time, soft real-time, general purpose distributed multimedia, scientific computing, etc.). Finally, while the discussion here was motivated by real-time constraints, this combination of admission control, resource reservation, and reflection is valuable even for systems without real-time requirements.

# References


[1] T. Anderson, B. Bershad, E. Lazowska, and H. Levy, Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism, *Proc. 13th ACM Symposium on OS Principles*, October 1991.

[2] D. Engler, M. F. Kaashoek, J. O'Toole, Exo-kernel: An Operating System Architecture for Application Level Resource Management, MIT TR, March 24, 1995.

[3] M. Jones, D. Rosu, and M-C Rosu, CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities,

*Proc. Sixteenth ACM Symposium on Operating Systems Principles*, Oct. 1997.

[4] B. Marsh, M. Scott, T. LeBlanc, and E. Markatos, First Class User Level Threads, *Proc. 13th ACM Symposium on OS Principles*, October 1991.

[5] K. Ramamritham, and J. Stankovic, Scheduling Strategies Adopted in Spring: An Overview, chapter in *Foundations of Real-Time Computing: Scheduling and Resource Management*, edited by Andre van Tilborg and Gary Koob, Kluwer Academic Publishers, pp. 277-306, 1991.

[6] K. Ramamritham, J. Stankovic, and P. Shiah, Efficient Scheduling Algorithms For Real-Time Multiprocessor Systems, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 2, pp. 184-194, April 1990.

[7] C. Shen, K. Ramamritham, and J. Stankovic, Resource Reclaiming in Real-Time, *Real-Time Systems Symposium*, December 1990.

[8] J. Stankovic and K. Ramamritham, The Spring Kernel: A New Paradigm for Real-Time Operating Systems, Special Issue of *ACM OS Review*, Vol. 23, No. 3, pp. 54-71, July 1989.

[9] J. Stankovic and K. Ramamritham, The Spring Kernel: A New Paradigm for Real-Time Systems, *IEEE Software*, Vol. 8, No. 3, pp. 62-72, May 1991; also in text on *Readings in Real-Time Systems*, IEEE Press, editor Mani Krishna.

[10] J. Stankovic, On the Reflective Nature of the Spring Kernel, invited paper, *Proc. Process Control Systems '91*, Springer Verlag, Berlin, pp. 2-18, February 1991.

[11] Y. Yokote, The Apertos Reflective Operating System, *Proceedings of Object-oriented Programming Systems, Languages, and Applications*, ACM Press, October 1992.