Resource Partitioning in General Purpose Operating Systems

Experimental Results in Windows NT

D. G. Waddington and D. Hutchison
Distributed Multimedia Research Group,
Computing Department,
Lancaster University,
Lancaster LA1 4YR, UK

e-mail: [dan, dh]@comp.lancs.ac.uk

ABSTRACT

The principal role of the operating system is that of resource management. Its task is to present a set of appropriate services to the applications and users it supports. Traditionally, general-purpose operating systems, including Windows NT, federate resource sharing in a fair manner, with the predominant goal of efficient resource utilisation. As a result the chosen scheduling algorithms are not suited to applications that have stringent Quality-of-Service (QoS) and resource management requirements, such as continuous media streaming applications. Several approaches to resource management that are able to meet these requirements, including the use of deterministic schedulers, are well established. Operating systems already exist that provide services for fine-grained resource partitioning and QoS based allocations. Nevertheless, the question arises: can we introduce suitable management mechanisms into general-purpose operating systems that will provide adequate support for the majority of resource sensitive applications, particularly multimedia? Or are there inherent problems in general-purpose operating system architectures that demand significant re-engineering?

We attempt to answer this question by proposing the Protected Virtual Machine (PVM) architecture. This introduces mechanisms into Windows NT that enable partitioning of resources and protection from resource interference. The functionality of the PVM architecture is two-fold. First it introduces the necessary scheduling mechanisms and atomic resource abstractions into the operating system, and second it provides a higher level reservation and admission service directly to applications. The resulting system is able to provide both a best-effort service for applications with less stringent needs and a guaranteed service for multimedia and other resource critical processing in a shared environment. By enhancing the existing operating system, legacy applications can still be supported, thus avoiding a prominent obstacle in existing real-time and multimedia operating systems.
1 Introduction

Enterprise level general-purpose operating systems, both workstation and server, are traditionally designed for efficient resource utilisation. Applications share a common set of resources, which are initially scheduled in a fair weighted manner. This approach results in both an efficient and fair division of resources to applications. However, certain applications, in particularly multimedia processing applications, are inherently sensitive to resource availability. Therefore, the use of priority-based federation policies often results in these applications failing to meet their processing goals, and thus rendering the application ineffective. To successfully provide for both resource sensitive and insensitive applications, within the same operating system environment, appropriate resource management and scheduling algorithms must be introduced. To do this one can either completely re-engineer the operating system with QoS sensitive applications in mind, or alternatively extend an existing architecture with appropriate mechanisms. The former approach is likely to provide better results, but is however severely constrained by its inability to support legacy applications and software technologies.

In this paper, we introduce a number of resource management mechanisms into Windows NT, in an attempt to facilitate resource partitioning for resource sensitive applications. These mechanisms collectively define the Protected Virtual Machine (PVM) architecture, which presents each application with a virtual set of resources, and associated service class that defines whether access to the resource is ‘best-effort’ or ‘guaranteed’. The virtual set also protects guaranteed resources, making them uniquely available to the application.

The notion of service classes has existed in the networking community for some time, differentiating between traffic or service types and enabling users to assign network resources according to defined end-to-end Quality-of-Service (QoS) requirements. For example, within the ATM networking community [2], the application is assigned a particular QoS class that is suited to the nature of the application’s network communications. Each QoS class maps down onto a set of parameters for one of five different traffic classes, Unspecified Bite Rate (UBR), Constant Bit Rate (CBR), non real-time and real-time Variable Bit Rate (VBR), or Available Bit Rate (ABR). The choice of traffic class, and associated parameters, dictates the cell switching resources in the network. We believe that approaches to resource management currently found in networks can also be applied, to some extent, to the end-system.

Our work draws a parallel to the virtual machine support in the IBM MVS architecture, later subsumed by the IBM Enterprise Systems Architecture [9]. IBM/ESA is a mainframe system that provides the customisable division of processor and storage between multiple applications. In terms of smaller-scale operating systems, much valuable research on specialised support for resource sensitive processing, such as demanded by multimedia applications, has also been carried out [4,7, 13-17]. These approaches are generally based upon the design of an operating system from scratch, rather than the evolution of existing general-purpose operating systems already deployed in enterprise environments. Often the reason for developing a specialised platform is due to the difficulties in integrating required functionality at the lowest level. For example, exchanging a priority based scheduler with a more appropriate scheme, does require the
availability of source code and considerations as to the effect of the transplant. In our own work we have both designed the replacement resource scheduling mechanisms so as not to effect legacy applications and put the mechanisms into place through access to the operating system source code.

The development of the PVM is being carried out within the context of the British Telecom - University Research Initiative (BT-URI) on the Management of Multiservice Networks (MMN) assisted by source code licensing from Microsoft. Lancaster University's part in the URI project is to investigate end-system QoS management through the design and development of a component-based Distributed Resource Management Architecture (DRMA) [22] which, is a platform for the deployment of multimedia services over broadband networks. The architecture incorporates QoS-based distributed bindings in conjunction with multimedia operating system services, assuring end-to-end QoS through resource monitoring, management and adaptation; the PVM is designed to facilitate the necessary resource management. We also expect that the PVM architecture will be useful in server end-systems where a minimum level of service is to be sustained across individual server applications. An example scenario is the division of resources across Internet services (ftp, http, a/v streaming) whereby the system administrator wishes to avoid the over consumption of resources by a particular service, resulting in the starvation of others.

In section 2 we present the motivation for and design of the PVM and discuss its relevance to the end-to-end provision of QoS. Then, in section 3 we discuss the requirements for the scheduling of each of the resource sets in the end-system. We also discuss our approach to implementation and issues surrounding the deployment of the resource modules and their associated schedulers in the Windows NT operating system. In section 4 we evaluate the system and offer the results from our prototype implementation, then in section 5, we draw some conclusions. In section 6, we outline some important related work, and then finally in section 7, we discuss the direction our future work in this area will take.

2 The Protected Virtual Machine and QoS Management

The role of the PVM, as the name suggests, is to provide applications with a set of protected endsystem resources. These represent a ‘virtual machine’ environment that is an encapsulation (as a subset rather than a superset) of applications and their physical resources. This results in the ability to manage resources in a shared end-system and make allocations according to individual application requirements. An important aspect of our approach is that conformant applications that are willing to undergo reservation are allocated resources for sole use. Non-conformant legacy applications are free to use whatever resources are left in the end-system, under a best-effort service. Furthermore, the PVM may be used externally by a third party process (maybe a management process) and does not necessarily need to be invoked by the client directly. This means that legacy applications can be ‘inserted’ into virtual machines without the need to adjust existing code. In order to construct a PVM resource set, we must co-ordinate the complete set of reservations on an individual basis. End-systems generally consist of a wide array of resources
each that must each be managed to achieve a particular task. The PVM is currently being incorporated in a QoS management framework, and is responsible for making resource reservations according to requirements that arise from the QoS mapping process.

2.1 Scheduling Services through Static Scheduling Algorithms

The PVM federates resources based on one of two services, ‘best-effort’ and ‘guaranteed’. The guaranteed service provides controlled access to resources according to resource reservations that are made \textit{a priori}. Of course, this does mandate that an application knows what its requirements are beforehand, so that the necessary reservations can be made (this is usually determined by running the application through a monitoring tool). Furthermore, an application cannot simply overbook resources because they are costly (in terms of some value of availability) and QoS verification processes are used to ensure that the application is making reasonable use of the reservations it has made. Within the QoS management framework, requirements are estimated for a given application and then QoS adaptation is used to adjust to inaccuracies in the forward reservation \cite{23}. The guaranteed service, as defined in the PVM end-system, is subtly different for other research efforts in the support of real-time resource scheduling for multimedia applications. Many real-time systems use dynamic scheduling to schedule resources in line with temporal requirements \cite{4, 6, 11}. Algorithms adopted in these systems, such as Earliest Deadline First (EDF) and Weighted Fair Queuing, are dynamic algorithms that do not require prior reservation of resources, although it is often possible to carry out statistical admission testing. One might reason that these algorithms provide more efficient resource utilisation, is generally more effective than other static scheduling schemes and do not demand that requirements be known beforehand. However, with respect to multimedia processing, we offer the following reasons for our own choice of static scheduling techniques within the Windows NT PVM architecture:

- Reservation and admission control means that resource usage saturation can be prevented and therefore access to resources can be properly protected. The performance of dynamic scheduling algorithms can become degraded during overloaded conditions and admission testing is more difficult.

- The complexity of static scheduling algorithms occurs at reservation time and therefore does not require any run-time processing. Thus, the complexity of the scheduling mechanisms is reduced.

- Within multimedia systems, the granularity of resource scheduling is relatively lax (soft real-time). This is generally due to buffered processing. Consequently, there is no need for the determination of exact requirements, which is not reasonably possible \textit{a priori}.

- Static scheduling techniques are inherently deterministic.
Continuous media processing is generally periodic rather than sporadic and therefore, determination of resource requirements beforehand is more viable.

The alternative to the guaranteed service is ‘best-effort’. The existence of this service is vital to furthering overall efficient resource utilisation. It is analogous to the best-effort services in the networking community such as the ATM Available Bit-Rate (ABR) service. This class provides fair sharing access to the resources that are not being consumed by the guaranteed service. Thus, superfluous resource availability arising from poor resource utilisation is absorbed into the best-effort service and made available to other applications.

2.2 Extensible Resource Abstractions and Split-level Scheduling Scheme

End-system resources are characteristically different. They present services through different abstractions or interfaces. In the PVM reservation scheme, the same resource abstraction is used across all end-system resources. All resources are encapsulated as Microsoft Component Object Model (COM) objects, and provide a reservation service via a strictly defined resource management interface, and a utilisation service via the basic operating system API. This means that in terms of the necessary management processes, i.e. reservation and admission, each resource within the PVM end-system can be treated polymorphically. This approach means that the system can be easily extended to incorporate new types of end-system resource into the reservation scheme. The resource management interface, namely IResource, which is provided by each individual resource module, is not directly accessible by the application. Instead it is only made available to a higher level scheduler (termed the primary scheduler) in a split-level scheduling scheme.

![Resource Abstractions in the PVM](image)

The primary scheduler and resource modules manage a large portion of the scheduling complexity. These are not involved during resource utilisation, only at the point of admission. Their aim is to translate coarse-grained (application level) resource requirements into a set of finer-grained scheduling requirements, and to manage the instantiation and cancellation of reservations over a period of time. Resource modules are instantiated dynamically as required, according to the requirements passed into the primary scheduler. This enables the scheduling policy for a resource to be loaded according to specific application requirements. However, if multiple instances of the resource components exist concurrently, they must all share the same secondary scheduler and maintain co-ordination accordingly. Jones et al., in their Vassal work.
have devised a scheme for dynamically loading low level (equivalent to secondary) processor scheduling policies in Windows NT [5]. However, this approach uses a hierarchical construction, and thus interference between multiple scheduling policies is more difficult to control.

In line with our fuller QoS management framework [22], we are using XML [25] to specify both QoS requirements and resource requirements (in terms of the PVM we are only interested in the latter). This technique provides both a powerful and open approach to specification, which can be easily extended by an administrator or developer to meet new and potentially unforeseen requirements. Thus, reservations that are made to the primary scheduler are described directly in terms of XML or as an identifier to an XML class description. The XML description is translated into resource specific metrics that are then forwarded onto the individual resource modules. The resource modules then translate metrics into secondary scheduling requirements. Using XML means that the format of the specifications may be standardised through document templates. An example XML resource class specification is given below:

```xml
<?xml version="1.0"?>
<!DOCTYPE RESOURCE_CLASS SYSTEM "resource.dtd">
<RESOURCE_CLASS identifier="F52E013-7AED-11D2-ABE1-002048082D4C">
  <RESOURCE>
    <NAME>Processor</NAME>
    <INSTANCE>0</INSTANCE>
    <VALUE units="signature">XX0000XX0000</VALUE>
  </RESOURCE>
  <RESOURCE>
    <NAME>PhysicalDisk</NAME>
    <INSTANCE>2</INSTANCE>
    <VALUE units="percentage" qualifier="min">10.0</VALUE>
  </RESOURCE>
</RESOURCE_CLASS>
```

The example above uses percentiles to describe resource requirements for the disk resource. Of course, this metric is relative to the system on which the reservation is made. Therefore, we assume that the XML class descriptions are tailored given end-system capabilities. Furthermore, there is no reason why absolute metrics cannot be used, provided the secondary scheduler is capable of interpreting it appropriately.

3 Scheduling Resources

In this section we present the design and implementation of the PVM in the Windows NT operating system and enforce our rationale for the choice of scheduling algorithms. We believe that the changes we have made in the prototype implementation can be suitably replicated in any general-purpose operating system. The current implementation provides three core resource
modules; the processor, disk and memory. We hope to extend this later to potentially include specialised multimedia processing devices (such as MPEG-II codecs) and network devices which are particularly important in distributed multimedia systems.

The primary scheduler, which provides the front end of the PVM service, is engineered as a Distributed COM software component that is integrated into an End-system Resource Management Service offering a richer set of resource management services. As previously discussed, the primary scheduler is responsible for mapping/ translating application level requirements into a minimal scheduling policy that can be readily interpreted at a lower level by the secondary schedulers. To achieve this, the primary scheduler maintains a representation of the signature that is currently being used by the secondary schedulers. This also allows scheduling decisions to be made with respect to existing reservations and eliminates the need for continual state querying at a lower level. It is also important to note that access to the resource modules is carefully controlled (via security protocols) and that the primary scheduler is the only entity that can directly access these. This prevents applications bypassing the resource manager and accessing the resource modules directly and thus potentially jeopardising system integrity. Finally, the resource modules are implemented into the operating system kernel and co-operate with user-level policy engines (used to interpret reservations) and the reservation API in the form of the IResource interface.

3.1 Processor

Probably the most important resource module in the PVM architecture is the processor resource module. The processor is a resource required by all applications and is therefore under continual contention. Although much research has already been carried out on processor scheduling algorithms and many efficient algorithms already exist, we have designed our own very simplistic scheduler that can be introduced into an existing operating system without extensive alterations to existing code. The novelty in our approach is exploitation of both 'priority' and 'period'. The allocation of priority federates the timeliness of resource sharing between threads in a single guaranteed process or across all thread in the best-effort service. Allocation of time (period) indicates the duration of access to the processor. These two characteristics together determine overall processor QoS. Therefore our scheduler is based on the combination of priority based scheduling (as found in many traditional general-purpose operating systems) and a more deterministic period assignment scheme. The schedulable client is currently defined as a thread belonging to a given process; the resource is shared between associated threads. The application programmer is responsible for assigning priorities for intra-process thread scheduling. Furthermore, priority based scheduling is also used to schedule threads serviced under the best-effort class, allowing application programmers to assign respective priorities for best-effort threads. Consequently, priorities can also be used for time-critical operating system services in order to ensure system critical tasks are performed providing sufficient resources exist.

Within the PVM architecture, the processor resource is scheduled at the lowest level by the Processor Secondary Scheduler (PSS). It is activated irrespective of the priority level of the
Currently executing thread. In terms of engineering, the PSS is attached to the quantum end and thread ready scheduling points, see Figure 2. The scheme works by searching all of the dispatch queues for ‘guaranteed processes’ and dispatching the appropriate threads if they are ready for execution. A similar scheme has been used by [15] except that this work uses a rate monotonic slot allocation. The slot is incremented consistently every scheduler interrupt interval (10ms). If no threads belonging to the scheduled client exist in the ready queues, then the time slice is given to the best-effort service. However, before an increment is performed, providing free slots are available, additional 'chances' are assigned by temporarily shifting the reservation forward into a free slot by up to a maximum distance defined according to the scheduling period. These reservations are only temporary and are cleared once the current indicator has passed. If a dynamic slot shift cannot be performed then the slot is incremented and the process looses its chance at using its reservation.

If a process revokes its time-slice, then the reset of the quantum is assigned to the best-effort service. This avoids the reservation time line becoming squashed, and thus skewing future slot reservations. If a given slot does hold a reservation, or the client is unavailable, then the time-slice is also given to the best-effort service and allocated according to the priority-based scheduling scheme. Note also that a guaranteed service thread that becomes ready may also pre-empt a best-effort thread.

![Fig. 2. Adjusted Process Scheduling Flow](image-url)

The PSS maintains a time-slot scheduling signature; each represents a unit of access to the processor, i.e. a time-slice (in our NT-based implementation we have adjusted this to 10ms on an Intel 80x86 platform). Each slot holds a single reservation that is made up of process identifier, client identifier, and an indication of tolerance. Tolerance represents the dynamic flexibility of

---

1 It should be noted that 10ms represents the scheduler interrupt rate and is in no way connected with finer grained multimedia timers.
the reservation with respect to time. This flexibility, either backward (BFlex) or forward (FFlex) indicates the number of slots forward or backward a reservation can be moved, see Figure 3. The purpose of this is to enlarge the number of concurrent reservations in the PVM. Reservation requests are initially translated by the primary scheduler into secondary scheduling signatures. These translations may reduce the granularity of the requirements description, e.g. from percentile to periodic definition. If the signature cannot be admitted without change, then the signature to be admitted is allowed to flex. This still may not be sufficient to carry out the admission, in which case the existing reservations that conflict with the new signature are flexed as required. If after attempts to flex both the new signature and the existing signature still result in conflicts, then the admission is rejected.

![Figure 3. PVM Processor Scheduling](image)

Another important aspect of the PSS is in its handling of the best effort service. Of course if a guaranteed thread which is scheduled for execution by the PSS is not on the ready queue, then the time-slice is subsumed by the best-effort service. In this event, the PSS uses the existing priority based scheduler to decide which thread to execute next. Furthermore, in order to maintain a minimum best-effort service, the PSS scheduling signature reserves a number of time slots for the best-effort service. In the current prototype, entering the best-effort lottery is permissible by any thread in the system, which means that in fact the guaranteed service represents an ‘at least’ service. It is perfectly valid for a guaranteed service thread to receive more resources than actually reserved. The final aspect of interest concerning the PVM processor scheduler, is that the scheme also permits the specification of a potentially complex periodic access signature.

### 3.2 Disk I/O

Controlling access to storage resources is particularly important in server-side end-systems and lack of I/O protection in current server architectures, often results in complete system degradation and even priority inversion for processor reservations. As with other resources, the disk resource module consists of kernel mechanisms that interact directly with the PVM primary scheduler together with a user-level service interface. In selecting a suitable scheduling
algorithm for the disk I/O secondary scheduler, we examined the characteristics of the resource. Compared to the processor resource, device communications are relatively coarse grained and sporadic tasks. Factors that cause the sporadic nature of disk I/O include location of the information on the disk relative to the current head position, scattering of the data in non-contiguous blocks and general disk device performance parameters; all these contribute towards non-deterministic service times. It could be argued that multimedia applications should generally exploit data placement schemes, resulting in more periodic access pattern. However, we are making the assumption that the storage is shared among different types of application, making such an approach infeasible.

Another characteristic of storage device I/O, relevant to the choice of scheduling algorithm, is the inability to pre-empt servicing tasks. In the Windows NT operating system, the unit of disk processing is the I/O Request Packet (IRP). Each IRP is an encapsulated request that defines an I/O task, including the type of operation to be carried out (read or write), the number of bytes to manipulate, the synchrony of the request etc. Once an IRP had been submitted for servicing, it generally cannot be interrupted and must be left until completion. Unlike a processor time-slice, the length of time required to service an IRP cannot be readily determined and therefore applying the same scheduling algorithm as adopted for the processor resource is not an ideal solution. Instead, we have deployed a scheduling scheme based upon ‘Lottery Scheduling’ [24]. The principal reason for this choice is that it is both an alternative to the processor secondary scheduling scheme and that it is neatly suited to our needs. This scheduling is not fully deterministic, but is sufficiently effective for our own purposes and very straightforward to implement. Lottery scheduling is quite unusual in that it is, in strict terms, a dynamic scheduling algorithm, but retains the reservation/admission characteristics of traditional static algorithms. It efficiently implements proportional-share resource management and is probabilistically fair. Lottery scheduling consists of the distribution of access ‘tickets’ according to an allocated share. A randomised number generator is used to determine the winning ticket, resulting in the holder being granted access to the resource. Ticket management can be quite complex, allowing ticket transferral and the incorporation of ticket cost. Although lottery scheduling does not allow the specification of complex access patterns, it is surprisingly efficient and particularly appropriate for disk I/O management.

We have engineered this scheduler as a kernel-mode layered device driver. The module sits between the file system driver (e.g. FAT, NTFS) and the physical device driver (e.g. ATAPI). Only IRPs that are not associated with virtual memory paging and other system-based accesses are forwarded to the scheduler. IRPs that do qualify are placed on a wait array according to the client identifier. The wait array is asynchronously serviced by a worker thread that is continuously running the lottery and servicing the IRP for the respective winning process. When there is no ticket holder or the ticket holder does not have an IRP ready for servicing, then an alternative IRP under the best-effort service is scheduled via a cyclic selection scheme.
Again, in line with the processor scheduler, a minimum level of service is maintained for the best-effort class. This is achieved by reserving a small proportion of tickets for the best-effort service. We have also had to eliminate IRP queuing within the lower-level physical device driver, since the effect of retaining these queues is that the intermediate wait queues are flushed immediately and therefore the scheduling becomes ineffective. Finally, we have introduced performance counters into the secondary scheduler in order to determine more accurately the outcome of the statistical lottery and also verify that reservations are being used. These counters can be used to determine actual access to the disk resource on a per client basis.

3.3 Memory

Our last prototype PVM module is the memory resource module. Management of memory resources is a subtly different task, since the duration of access is relatively long in comparison to that of the processor and disk resources. Most general-purpose operating systems use a Virtual Memory Management (VMM) scheme that allows the allocation of memory beyond physical capacity with modern management features such as mapped files and copy-on-write. Virtual memory management is a complex topic and beyond the scope of this paper. However, in order to refresh the reader’s memory, we shall describe the scheme in brief and provide sufficient context for the memory management mechanisms placed in the PVM.

Each process maintains its own unique virtual address space, which is a set of memory addresses available for threads to use. This address space is generally much larger than the physical memory of the system. When a process references its virtual memory, it does so with a 32-bit pointer (at least within 32 bit operating systems) that is used by the Virtual Memory Manager.
(VMM) to determine the location in physical memory to which the virtual address maps, see figure 5. However, some portions of memory may actually reside on disk or backing store. This means that data or code that is infrequently accessed is held on backing store thus retaining physical memory for better use. To facilitate this scheme, memory is managed in fixed size blocks known as pages, which are usually 4Kb or 8Kb. If a process references an address whose page is not in physical memory, then a page fault occurs. This triggers a process known as paging that locates the faulted page within the page file and then loads the appropriate page into physical memory. If there is insufficient space in physical memory, a page must be first swapped out and written to the page file. However, each process keeps a minimum number of pages locked into physical memory known as the working set. Different variations of this scheme exist in different operating system architectures. Variations include a number of different policies, which are used to determine when pages should be swapped in, which pages are selected for swapping out, and how virtual addresses are translated to physical addresses.

**Fig. 5. Basic Virtual Memory Management**

General-purpose operating systems usually protect memory in three forms. First, the hardware prevents threads from accessing address spaces of other processes. Second, a distinction is made between the mode of operation, either kernel-mode allowing threads to access system code and data, or user-mode which doesn’t. And third, page-based protection mechanisms maintain a set of flags for each memory page that determines the type of access permitted. However, these mechanisms are aimed at protecting a process’s memory resources from unwanted interference by other unauthorised processes in the system. They do not prevent a process from consuming more than its reasonable share of memory resources, one of the original goals of the PVM.

We cannot enforce a quota limit on all non-conformant processes in the end-system, as this conflicts with our original requirement of providing a best-effort service. However, within the Windows NT operating system, memory resources are managed via a set of quota variables that are associated with the end-system as a whole and with each individual process. Quotas are managed by the VMM dynamically and are adjusted according to both resource demands and availability. We have identified two characteristics of virtual memory, which can be regarded as
manageable units. The first is *space within the page file*. When a process commits memory, the VMM reserves a portion of the page file for pages that are not resident in physical memory (termed invalid pages). Generally the page file is dynamically expansible and its size is adjusted in line with demand. However, there is usually a threshold to the amount the page file may be increased, thus effectively constraining the total amount of virtual memory the system can provide. The maximum amount of page file assigned to an individual process is governed by its page file quota; one of many quotas associated to an NT process. Adjustment of the page file quota is relatively straightforward providing that there is sufficient page file space at the point of reservation.

The second characteristic, which more prominently affects application performance, is the number of pages a process can have concurrently locked in physical memory, i.e. the *size of the process's working set*. To describe our approach to working set size control, we must briefly describe the working set management scheme in Windows NT. On creation, each process is assigned two thresholds; a minimum and maximum working-set size. The minimum defines the smallest number of pages the VMM attempts to keep locked concurrently in physical memory, whilst the maximum defines an expansion range that can be used if the process is causing a considerable number of page faults. If the working set is too small, then the process incurs a large number of paging operations and thus a substantial overhead is incurred through continuously swapping pages to and from the backing store. Alternatively, if the working set is too large, few page faults occur but the physical memory may be holding code/data that is infrequently referenced and thus overall efficiency reduced. In Windows NT, the Memory Manager (MM) adjusts the working sets once every second, in response to page-in operations or when available memory drops to below a given threshold. If free memory is plentiful, the MM removes infrequently referenced pages from working sets whose current size is above a given minimum, this is known as *aggressive* trimming. However, if free memory is scarce, the MM can *force* trimming of pages from any process until it creates an adequate number of pages, even beyond the minimum threshold. Of course, processes that have extended working sets are trimmed in preference to those that have the minimum working set size.

In our prototype implementation, a reservation to the PVM memory module for physical non-paged memory is translated into the appropriate expansion of the process's working set size, particularly the minimum threshold. The minimum working set size regulates the number of pages a process can concurrently lock (or pin) into physical memory. Therefore, an application that wishes to reserve a portion of memory, makes the reservation request to the PVM which then increases the working set thresholds accordingly. It is then the responsibility of the application to pin the pages into physical memory as required. The pinning process is arbitrated on a first-come first-serve basis. If insufficient physical pages are available, the operating system will reject the lock request. Consequently, the application is expected to lock its reserved pages into physical memory soon after allocation. If there are insufficient free physical memory resources, the PVM attempts to force the best-effort processes to write pages back to file store until sufficient free physical pages are available. If insufficient space can be made, then the reservation is rejected. As with the other resource modules, a portion of the total physical memory is reserved for a
minimal best-effort service. This is achieved by maintaining a count of the total number of physical pages being used under the guaranteed service and checking this against a portion of the total physical page capacity. It should also be noted that the best-effort service is strictly a minimal guaranteed service defined by the system assigned minimum working set size. Additional services can be gained through expansion into unreserved memory, although this may be forcefully revoked later. It is also assumed that processes cannot bypass the PVM and directly alter their own working set size. Finally, special consideration is made for pages that are being used for shared memory purposes. If any client of a shared page has undergone PVM reservation then the page is classified as reserved even though other clients may not have made a reservation.

4 Evaluation

In this section, we present results taken from a Windows NT prototype PVM implementation. The results were taken from an Intel Pentium II-based Windows NT Server 4.0 platform. Measurements were extracted via a performance monitoring utility that reads the system’s low-level extensible performance counters. Because of the temporal granularity and deterministic nature of memory resources, we have not included measurements from the memory module. The first results describe measurements for processor and disk utilisation from a scenario consisting of a single end-system with multiple processes, each competing for the same resource. Initially all processes accessed resources on a best-effort basis. Incremental reservations were made thereafter. Measurements were taken over a period of 30 seconds and then an average service rate for the guaranteed service process was measured for that period. These results were used to determine ‘effectiveness’ of the PVM, by determining what the PVM service should theoretically provide for a given resource reservation and comparing this value with an the observed level of service derived from monitoring results. In both cases, the theoretical level of service is calculated as follows, where $R$ is resource and $SR$ is the service rate:

$$SR_{\text{guaranteed}} = \left(\frac{R_{\text{allocated}}}{R_{\text{total}}} \times SR_{\text{max}}\right)$$

$$SR_{\text{best-effort}} = \left[\frac{SR_{\text{max}} - SR_{\text{guaranteed}}}{n}\right]$$

$$SR_{\text{expected}} = SR_{\text{guaranteed}} + SR_{\text{best-effort}}$$

Graphs 1 and 2 show results taken from the processor and disk modules respectively. Both graphs show an apparent collation between the measured service and the theoretical expected level of service, indicating reasonable effectiveness of the PVM scheduling mechanisms in this case.
The results also indicate that the measured service is generally slightly lower than the expected service, by around 5%. This effect is caused by the combined overhead of the second level scheduling mechanisms and operating system services that execute in privileged mode. It has also be suggested by Jones et. al. that this may be caused by problematic video device drivers causing interrupt stalls [12].

Graph 1. Results from Processor Resource Module

Graph 2. Results from Disk Resource Module

Graphs 1 and 2 are based upon service averages and therefore it is difficult to ascertain the exact fluctuations (or jitter) in the guaranteed service. Graph 3 illustrates the relationship between measured processor service and incremental changes in resource assignment over time. The graph also includes a measure of the best-effort service in the system. As you can see from the graph, each time more resources are assigned to the guaranteed service process (1 in 8 slot increments) the best-effort service is reduced. An appraisal of the results indicates that the guaranteed service fluctuates within about a 10% region of the reserved datum. These fluctuations are caused by both the granularity of the monitoring (performed at one second intervals) and the granularity of the reservation. For example, a reservation for 1 in 2 slots will be substantially smoother than a reservation for 1 in 32 time-slots.
Graph 3. Results from Processor Resource Module - Incremental Resource Assignment

The results we have obtained so far from the prototype PVM are quite promising. Nevertheless, the experimental scenarios are designed to avoid the effects of resource interdependencies. Each test process makes continuous accesses to a single specific resource, i.e. through a busy loop (CPU-bound) or continuous non-cached file requests (I/O bound). This approach means that we can determine the effect of the secondary schedulers without concern to the processes blocking for I/O or blocking for processor time. In a more realistic management scenario, applications are likely to require guaranteed access to a complete set of resources, and thus reservations must be co-ordinated across all resources in the virtual set.

<table>
<thead>
<tr>
<th>Transition</th>
<th>Allocation</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>mplayer1 (\rightarrow) 90% disk tokens</td>
<td>i/o amplitude becomes more varied, average throughput is not affected</td>
</tr>
<tr>
<td>B</td>
<td>mplayer1 (\rightarrow) all CPU slots</td>
<td>an increase in peak i/o throughput and processor usage is observed; lower priority best-effort application (cpubusy.exe) drops in processor utilisation</td>
</tr>
<tr>
<td>C</td>
<td>revoke disk allocation</td>
<td>i/o seems unaffected</td>
</tr>
<tr>
<td>D</td>
<td>revoke CPU allocation</td>
<td>return to original state; (cpubusy.exe) increases as an effect</td>
</tr>
<tr>
<td>E</td>
<td>mplayer2 (\rightarrow) 1:2 CPU slots</td>
<td>1 in 2 processor slots is shown to be sufficient for maximum processor demand of mplayer2 (around 50%); again the best-effort service (cpubusy.exe) is degraded</td>
</tr>
<tr>
<td>F</td>
<td>mplayer2 (\rightarrow) 90% disk tokens</td>
<td>no effect observed</td>
</tr>
<tr>
<td>G</td>
<td>reset all allocations</td>
<td>return to normal</td>
</tr>
<tr>
<td>H</td>
<td>cpubusy (\rightarrow) all CPU slots diskbusy (\rightarrow) 90% disk tokens</td>
<td>cpubusy.exe becomes guaranteed and consumes ~98% of the processor time. diskbusy.exe is unable to use its disk i/o tokens with limited processing time.</td>
</tr>
<tr>
<td>I</td>
<td>reset all allocations</td>
<td>return to normal</td>
</tr>
</tbody>
</table>

Table 1. Allocations made for Processor and Disk Cross-Interference Tests
Graph 4 below provides results from a test designed to show the effect of cross-interference between resource types. Measurements of the overall processor utilisation and disk throughput were made across four concurrent applications, two audio/video playback applications, one CPU intensive application and one disk I/O intensive application. A series of reservation allocations were made, these are detailed in Table 1.

![Graph 4. Results from Processor and Disk Cross-Interference Tests](image)

These results illustrate a tight coupling between processor utilisation and effective disk throughput. Reservations on disk I/O are shown useful only when sufficient processor availability exists. Thus, reservations that are made for a particular task are only effective when appropriate reservations are made for all resources that are required to perform the task. By making reservation for all resources, blocking can be avoided.
5 Related Work

Over recent years, Carnegie Mellon University has made a considerable contribution to the research of resource management in multimedia operating systems. Much of their work has concentrated on the introduction of resource abstractions and management mechanisms into the Real-Time Mach operating system. Mercer et al. introduced the novel reserve abstraction of Processor Capacity Reserves [16] a concept later extended to other end-system resources in the form of resource kernels [19]. Resource kernels provide resource-centric services which, in turn, can be used to satisfy end-to-end QoS requirements. A QoS manager sitting on top of a resource kernel makes adaptive adjustments to resources allocated to applications. Aspects of the use of resource management to affect QoS requirements are also addressed [18]. Their experimental work is limited to only two resources, the processor and file system. Real-Time Mach is not considered a general-purpose operating system, which is the primary context for our own work.

Jones et al. at Microsoft Research have developed the Rialto operating system [11] which is an object based real-time kernel designed and built from scratch for research in real-time systems. Recent work by Jones et al. also includes the Vassal [5] loadable scheduler support, which enables processor-scheduling policies to be dynamically loaded, via an executive module, into the kernel. The Vassal work is particularly relevant since it is based on direct adaptations to Windows NT.

The Nemesis operating system [14] from the Cambridge Computer Laboratory, Cambridge University, is a research platform that has been developed from the ground up and is designed to provide QoS guarantees to applications. The operating system has novel vertical structuring in that the vast majority of functionality comprising an application can execute in a single process, or domain. In comparison to our own work, the Nemesis research is focused on the development of a new operating system, rather than the extension of an existing commercial system such as Windows NT. As a result, the Nemesis operating system is specifically aimed at the support for multimedia applications. The generalisation and suitability of their abstractions for other application domains is yet to be proved. However, Nemesis does use a similar split-level scheduling scheme and a period-based processor requirement specification. Current work on Nemesis is addressing support for legacy applications by introducing a POSIX programming subsystem to the architecture.

Another significant research contributions relevant to our own are the development of the Global Resource Management System (GRMS) from the Honeywell Technology Centre and work on resource scheduling in open environments at the University of Illinois at Urbana-Champaign [8, 17]. There work takes a similar path to our own in that each of the end-system resources is managed independently and resources allocated according to QoS requirements. Their UNIX-based QoS-aware resource management system and uses a C++ programming interface and proprietary object model.

Finally, Banga et. al. have been working on an operating system abstract known as resource containers [3]. This, in a manner similar to our own work, allows resource sets to be assigned to
given tasks. Their implementation is more counter-oriented than our own, and is based on modifications to Digital UNIX.

6 Conclusions

When drawing our conclusions, we must recall the original goal of the work; “to determine whether it is possible to introduce mechanisms into general-purpose operating systems that will provide adequate support for resource sensitive applications”. One would expect that an operating system developed from the ground up with this goal in mind would provide better results. Nevertheless, we have shown that a good level of resource partitioning can be achieved, which is able to protect resource sensitive applications such as required by multimedia processing. It is important to note that partitioning does not only protect resource sensitive applications from interference by other applications in the system, but also protects these from resource demanding applications so that a minimum level of service is maintained.

We have identified the following limitations in our mechanisms. Some of these are resolvable, however some cannot be addressed without severely re-engineering the operating system. The two principal problems that have arisen from introducing partitioning mechanisms in Windows NT are as follows:

- **Finite Processor Scheduling Granularity** - the current implementation uses a processor scheduling granularity of 10ms (Intel 80x86 architecture). Consequently, the number of guaranteed applications that can be reasonably admitted is constrained by this figure. For example, it is not possible to reserve resources for 10 applications that each require 5ms processing time in each 50ms, since in 50ms there are only 5 guaranteed service slots. In order to increase the number of resource sensitive applications that can be admitted, we must make the scheduling granularity finer grained and possibly variable (such as found in the Nemesis operating system [14]). Of course, if the scheduling interrupts are made more frequent, then an increased overhead from context swapping is also acquired. In Windows NT, the interrupt for scheduling (Irq8) could be made more frequent, but this would require the system clock code to be altered accordingly. Nevertheless, these adjustments are feasible.

- **Client/Server Dependencies** - Windows NT and many other general-purpose operating systems, are based upon a client/server model [20]. This means that processes are often reliant upon a number of services, each provisioned as individual processes in the system. In Windows NT, the majority of performance sensitive operating system components run in kernel mode, within the System process. Consequently, when an application is performing large amounts of memory management, or device I/O, then it becomes reliant upon kernel-mode threads. In severe cases, this can cause guarantee inversions when an application that has made a resource reservation becomes blocked whilst waiting for a server process to perform some work on its behalf. In the current PVM implementation, resources reserved for a particular client process cannot be transferred to server processes, including the System process. However, in Windows NT we still able to get reasonable results since work done in kernel-mode and other server processes is relatively limited (5% overall processor time is
unusual). To introduce a scheme that would allow reservations to be shared across client and server processes would require exact dependencies to be defined and forwarded to the secondary level schedulers.

Regardless of these reasonably noteworthy limitations, the introduction of specialised resource management mechanisms has proved to be adequate in scenarios that do not require a large number of protected processes and where server process dependencies are not excessive. Thus, the answer to our original question of whether we can introduce adequate resource partitioning into existing operating systems is somewhat dependent upon the scenario and your definition of ‘adequate’. The question may also arise: why not just use priority based scheduling? Our answer to this is that whilst priority can provided relative statistical guarantees, it is largely dependent upon system load. Furthermore, even if priority based scheduling could be coupled with admission testing, determinism still cannot be achieved.

Our work to date on the development of the PVM has concentrated on three of the core resource modules; processor, disk and memory. Of course, end-systems generally consist of resources other than these, including specialised multimedia decoders/encoders and network devices. Future work will examine the inclusion of these additional resources into the extensible resource management model.

We also propose to carry out further investigation into the effects of resource interdependencies and examine how we can manage the end-system resources co-operatively in order to achieve desired QoS. This involves the continued development of the primary scheduler and further research into the specification of end-system resource requirements in conjunction with the development of mapping mechanisms required to translate application level resource requirements into finer grained scheduling requirements.

7 Acknowledgements

We would like to acknowledge the kind support of BT Labs in funding this research under their Management of Multi-service Networks University Research Initiative. We would also like to acknowledge Microsoft Corporation for granting us their Windows NT source code licensing for this project.

References


12. M.B. Jones and J. Regehr, “The Problems You’re Having May Not Be the Problems You Think You’re Having: Results from a Latency Study of Windows NT”, *Proceedings of*


