A Resource Management Framework for Priority-Based Physical-Memory Allocation

Kingsley Cheung **Gernot Heiser**

School of Computer Science and Engineering
University of NSW, Sydney 2052, Australia,
{kcheung,gernot}@cse.unsw.edu.au

Abstract

Most multitasking operating systems support scheduling priorities in order to ensure that processor time is allocated to important or time-critical processes in preference to less important ones. Ideally this would prevent a low-priority process from slowing the execution of a high-priority one. In practice, strict prioritisation is undermined by a lack of suitable allocation policy for resources other than CPU time. For example, a low priority process may degrade the execution speed of a high-priority process by competing with it for physical memory. We present the design of a flexible resource management framework which prioritises memory allocation, and examine a prototype implementation for the Mungi single-address-space operating system.

1 Introduction

Resource management is one of the main responsibilities of an operating system. Processes on a system are generally in competition for resources such as processor time, physical memory or secondary memory (disk space), and it is the task of the operating system to allocate them according to some policy. In general, different policies are used for different kinds of resources.

On the one hand, disk space is usually managed with a quota system which limits the amount of space a user can occupy. Alternatively, *economic* models can be used, which associate a price with a resource, and users "buy" or "rent" space, involving some form of payment [Anderson et al., 1986, Mullender and Tanenbaum, 1986, Drexler and Miller, 1988, Heiser et al., 1998b].

On the other hand, processor time is allocated according to some priority scheme. Priorities can be *hard*, meaning that a process will only execute if no higher priority process is runnable, or *soft*, meaning that a process' priority influences the frequency or duration for which its process is allowed to execute. Hard priorities are required for time-critical (real-time) processes but can lead to starvation, which is why time-sharing systems generally use soft priorities. Priorities can be static or dynamic, e.g., the priority of a CPU-bound process may decay over time. An alternative to priorities are schemes which allocate a certain *share* of available processor time to processes or groups of processes [Larmouth, 1975, Kay and Lauder, 1988, Waldspurger and Weihl, 1994].

Ideally the processor allocation policy should exclusively determine whether and at which relative speed a particular process is executing. In particular, a process which, according to the policy, is not to execute at a particular time should not influence the execution of a process which should be executing according to the policy. This, however, is difficult to ensure, as the allocation of different types of resources is not independent of each other. Most modern operating systems do not implement prioritised allocation for resources such as physical memory and thus do not completely isolate processes from other processes competing for the same resources.

For example, a low priority process executing while a high priority process is blocked waiting for an event, may occupy a large amount of physical memory. The higher priority process, once it becomes runnable, may soon get blocked on a page fault, and as a result may execute more slowly than intended. As well, frequent faulting by processes can degrade the performance of other processes, due to contention for I/O bandwidth.

One suggested cause of these problems is the derivation of memory prioritisation from processor prioritisation schemes [Chapin, 1997]. Memory is implicitly prioritised by assuming that a higher priority process, which is allocated more processing time, will automatically gain a larger resident set by accessing pages more frequently. This strategy worked reasonably well with the slow processors of the past. The high CPU speeds of modern computers permit lower priority processes to reference many pages while higher priority processes block, leading to a loss of memory prioritisation. This is usually exacerbated by the use of global replacement algorithms that focus on total system throughput rather than the performance of important applications.

The major challenge in managing system resources is therefore to design a generic framework capable of managing a diverse set of resources. Mechanisms for prioritising resource allocation and isolating processes from resource contention are needed. Such a design should meet the following requirements:

Fairness: Other than dictated by system policies, or explicit user requests (within the limits of those policies), processes should be treated equally and no processes should starve. The isolation of entities from resource contention by other entities and the prioritisation of resource allocation is necessary to ensure fairness.

Simplicity: There should be a simple model of resource prioritisation. Excessive complexity can discourage users from applying the mechanism.

Flexibility: A resource management framework should be flexible and provide simple abstractions suitable for building more complex policies as needed. It should keep mechanism and policy separate.

Performance: Resource allocation and prioritisation should be efficient and management over-

heads small. Poor performance would discourage users from managing their resource usage.

Protection: A resource management framework should operate within the operating system's protection system. Entities may only affect other entities within the constraints of their imposed resource policies.

This paper describes how the above requirements have been addressed in the design of a new generic resource management framework and how this provides a mechanism for prioritising physical memory for the Mungi operating system. An overview of Mungi is provided in Section 2, describing fundamental Mungi abstractions and an existing economic model for managing secondary storage in the system. Section 3 surveys related research in resource management, particularly on approaches for prioritising physical memory. The design of a generic resource management framework is presented in Section 4, followed by details on the mechanism for prioritising physical memory allocation in Section 5. Experimental results for a prototype model are presented in Section 6.

2 Mungi

2.1 Fundamental Abstractions

Mungi [Heiser et al., 1998a] is a single-address-space operating system (SASOS), and as such executes all processes on all nodes in a single, large virtual address space. This address space contains all persistent and transient data, simplifying data sharing and persistent storage. All operations on this address space are associated with a small number of fundamental abstractions: *threads*, *protection domains*, *capabilities*, and *objects*.

Mungi *threads* are kernel-scheduled units of execution. They form a hierarchy defined by the parent-child relationship. In this hierarchy, threads may only kill descendant threads and may not survive their creator threads unless adopted by a higher ancestor.

Each Mungi thread executes in exactly one *protection domain* that defines the regions of virtual address space it may legally address. Since virtual address space is allocated in consecutive pages as *objects*, protection domains logically describe a set of objects and a corresponding set of access rights to these objects. These access rights are implemented as *password capabilities* [Anderson et al., 1986].

Mungi capabilities contain an object address, a password to protect it from forgery, and a set of access rights. They may be stored and passed around freely without system intervention and are registered in a global, distributed data structure called the *object table* (OT). When validating a capability, the system compares it with registered capabilities. If a match exists and the requested operation is compatible with the access mode, access is granted. For performance, validations are cached for each protection domain. Cached validations are flushed periodically to permit the revocation of access rights to objects.

2.2 Secondary Storage Management

Mungi has a secondary storage management model designed to control the proliferation of objects in the system [Heiser et al., 1998b]. This model is based on the *rent* scheme in the Monash Password Capability System [Anderson et al., 1986] and *bank accounts* from Amoeba [Mullender and Tanenbaum, 1986]. Its main objective is to ensure users do not starve or exploit others through excessive use.

2.2.1 Bank Accounts

Secondary storage is managed by charging rent for backing store usage through special objects called bank accounts. Accounts with money available for rent charging have a *financial* flag set in their OT entry. Object creation is only permitted if a valid capability to a financial bank account is supplied to the system. Checking this is efficient since OT entries are stored with cached validations.

2.2.2 Rent Collection

Actual accounting is performed by a user-level background thread called the *rent collector* at a reasonable frequency. Since rent collection is asynchronous to other system activities, all object operations are free of accounting overhead.

The rent collector has a capability to the OT. During rent collection it traverses the OT to charge bank accounts for backing store used by their objects. The rent charged is the product of the amount of backing store used, the current storage cost per page, and the elapsed time since the last rent collection. The inclusion of time in the charge permits collection at irregular times. Should a bank account be overdrawn, it is rendered non-financial and object creation with this account is prohibited once cached validations are flushed and updated.

Short-lived objects are charged only if they exist during rent collection. Unlike traditional disk quotas, this permits users to temporarily utilise idle resources. Rent collection could be scheduled at irregular intervals to prevent users from deliberately minimising storage usage during collection periods.

To adjust to high demand, the storage cost per page varies with utilisation. It remains constant during low utilisation and increases sharply when secondary storage becomes scarce, forcing users to free storage as they can no longer afford to pay.

2.2.3 Income and Taxation

To fund rent collection, bank accounts receive a *salary* from parent accounts, with an infinitely rich root account funding all top parent accounts. This hierarchy allows users to create additional accounts to manage different groups of objects. Salaries are paid by a *pay master* thread, which periodically deposits income scaled with time into each account.

To prevent users from accumulating excessive income and buying out the system, *taxation* is performed prior to salary deposition. This operation is again scaled with time to prevent excessive taxation during irregular salary depositions. Moreover, the amount of taxation is adjustable, permitting the construction of policies that consider resource usage history.

2.2.4 Bank Account Operations

To protect the integrity of bank accounts, users are only given *read* capabilities to their accounts, allowing them to read but not modify account data. All account modifications are performed through a trusted path mechanism called *protection domain extension* (PDX) [Vochteloo et al., 1996]. This mechanism allows the controlled extension of a thread's protection domain for the duration of a procedure call. When a PDX procedure for modifying account data is invoked with a valid read capability, the caller's protection domain is temporarily extended to include a write capability to the bank account. All

write capabilities to accounts are kept by a *bank manager* to ensure modifications are done only by the accounting software.

3 Related Work

This section surveys related research in resource management, particularly work involving physical memory.

3.1 UNIX

UNIX [Thompson and Ritchie, 1974] systems generally have little support for fairly allocating resources to groups of processes and provide only quota abstractions for individual processes. Abstractions for managing physical memory are usually per-process resident set size limits. As with quotas, limits are inflexible and physical memory can easily be under- or over-committed. Together with global replacement algorithms, these per-process mechanisms do not prioritise physical memory allocation nor do they isolate processes from resource contention.

3.2 Amoeba

The Amoeba [Mullender and Tanenbaum, 1986, Tanenbaum et al., 1986] economic resource management scheme uses *bank accounts* to pay for resources with *virtual money*. To permit different policies or subsystems, money is supported in possibly convertible or inconvertible currencies. Users own *individual* bank accounts; services own *business* bank accounts to manage client money.

When a system service is required, clients request a *bank server* to transfer money from their individual account to the business account of the service provider. Advanced payments are made to amortise the overhead of transfers. Quotas or rental models may then be employed to control resource usage. For example, a secondary storage quota scheme would debit money per block allocated. Credit is returned when blocks are freed. A rental model, in contrast, would charge clients for resources at a rate of λ money, per unit resource and unit time. Clients need to trust service providers, however, to provide the amount of resources paid for.

3.3 The *Share* Fair Share Scheduler

Possibly the earliest ideas of *proportional sharing* are by Larmouth [Larmouth, 1975], whose work recognised the need to schedule CPU fairly between *users* rather than processes. These ideas were the basis for the *Share* fair share scheduler [Kay and Lauder, 1988].

Share schedules CPU according to user *entitlements*, as defined by their *shares* and resource *usage history*. Shares indicate the proportion of resource to which a user is entitled. Usage history, a decayed measure of resource consumption, affects usage response. Process priorities are adjusted according to these parameters to share the CPU amongst users. Defining shares for groups of users is also possible, permitting groups to share a machine at an organisational level. To ensure each group receives its fair share, Share increases the *effective* share of active users with the shares of inactive users in the same group.

Unfortunately, Share imposes a fixed hierarchical policy structure of users and groups. It does not permit more complex policies or differently structured policies for different subsystems. Nor does Share provide a uniform interface to manipulate process scheduling, but supports UNIX nice semantics instead. Employing process shares would be better, since priorities are generally difficult to understand and do not vary linearly with resource rights.

3.4 Stealth

Stealth [Krueger and Chawla, 1991], an operating system for networked workstations, has a distributed scheduler to prioritise resource allocation. It aims to insulate workstation *owner* processes from *foreign* processes started at other workstations by executing foreign processes at a slower rate during resource contention.

Stealth implements prioritised memory allocation using a variant of the Mach 2.5 kernel memory manager. Two separate sets of active, inactive, and free page lists are maintained for owner and foreign processes. Pages are moved between lists to favour owner processes over foreign process. As well, to isolate owner processes from thrashing foreign processes, Stealth only responds to foreign process page faults if no owner process page faults are waiting. This permits full utilisation of networked resources with isolation for workstation owners.

However, although Stealth achieves its goal of owner-process insulation, it cannot be applied to multiple user systems requiring resource prioritisation between many groups of logical entities. In particular, its memory allocation scheme does not discriminate between more than two groups. There is basically no mechanism by which policies may be specified and entity resource allocations enforced.

3.5 Opal

Opal [Chase et al., 1994], also a SASOS, manages *segments*, sections of its single address space, through explicit reference counting. Reference counts reflect the number of entities "interested" in a resource, and resources are released only at zero counts. They are used by applications and facilities like runtime libraries and garbage collectors to allocate and release storage space.

To prevent abuse of reference counts, *resource groups* are employed for resource accounting. Capabilities to these groups are needed whenever a resource is created or a reference count is incremented. When a group is destroyed, the resources associated with the group are released. Each thread has a current resource group and implicitly passes a capability to this group on system calls.

Resource groups also provide control over application resource consumption. Groups can be nested into a hierarchical tree to allow finer resource accounting. A capability to a resource group permits a holder to create *subgroups* and delete any descendant subgroup. Accounting charges for a subgroup are passed to its parent and presented to users at the root, requiring them to limit resource usage with quota or billing models.

Opal's resource groups present a promising approach, although with some shortcomings. For one, each Opal thread belongs to only one resource group at any instance of time, namely its current group. This is inflexible if different types of resources are to be managed, since it restricts the management of all types of resources to the same structure. Resource specific structures would allow more flexibility in policy construction. Furthermore, the Opal approach to managing secondary storage by having users declare an explicit "interest" in objects, and using this as a basis for reference counting and garbage collection, is insufficient. It suffers from the same problems as traditional file systems, where persistent objects are entered into directories with human readable names.

An object of no "real" interest to entities can indefinitely remain referenced by a directory entry and not reclaimed by the garbage collector. We argue that users must manage storage [Heiser et al., 1998b]. The system can therefore only encourage users to clean up by charging users for resource usage. Since quota approaches are inflexible, this is best achieved by billing models.

3.6 Tickets and Currencies

[Waldspurger and Weihl, 1994, Waldspurger, 1995] describe proportional sharing using *tickets* and *currencies*. Tickets are essentially shares and define the proportion of resources a client may use. As mutually trusting clients can inflate their number of tickets, currencies are used to denominate tickets and provide isolation between clients. A *base* currency denominates the tickets directly proportional to the amount of resources allocated to clients and has defined exchange rates with *local* currencies.

To prioritise memory, two algorithms are introduced as possible methods for choosing a page replacement victim: *minimum-funding revocation* and *inverse lottery scheduling*. The former algorithm revokes a frame from the client with the least tickets per frame; the latter uses a pseudo-random number generator to select a victim, with the probability a client is chosen being inversely proportional to its number of tickets. Using a tree data structure, $O(log(n))$ implementations are possible for both algorithms, where n is the number of clients in the system.

Unfortunately, this framework has shortcomings. No consideration is given to accounting for shared frames. Memory sharing is important to the performance of modern systems. Copy-on-write sharing avoids copying overhead, shared libraries reduce startup latency and improve memory utilisation, and write-shared pages are the most efficient interprocessor communication mechanism.

Furthermore, if entities frequently enter or leave the system, frequent and possibly expensive changes in exchange rates occur. Moreover, when entities enter or leave the system, the inflation or deflation of tickets proportionally adjusts the resource rights of entities in the same currency. Policy, not the framework, should determine how tickets are adjusted among entities. These issues have yet to be considered, although other improvements, such as the removal of the upper and lower limits imposed by currencies [Sullivan and Seltzer, 2000], have been made by subsequent research.

3.7 IRIX

[Verghese et al., 1998] presents a performance isolation model for managing resources in the IRIX operating system. It partitions resources into isolated units called *Software Performance Units* (SPUs). Processes in the same SPU contend for resources allocated to the unit but do not experience any performance degradation from processes in other SPUs.

To partition physical memory frames, each SPU has an *entitled*, *allowed*, and *used* page count. The entitled page count represents the share of pages a unit is permitted, whereas the allowed page count serves as an upper limit. Used pages include process pages and kernel pages used on behalf of the unit.

For improved throughput, idle memory is loaned between SPUs by periodically adjusting allowed page counts. Free pages are distributed to SPUs under memory pressure and revoked when loaning SPUs require the lent resources. Since this revocation may involve writing dirty pages to disk, a set of free pages are reserved to minimise the revocation time.

The isolation model also accounts for shared resources, including shared frames. When a page is first accessed, it is assigned the SPU identifier of its faulting process. Subsequent access by a process in another SPU will assign the page to a default *shared* SPU. This method, nonetheless, does not fairly charge for shared frames since it effectively charges shared frames to all units. It allows heavy sharing to occur between several SPUs at the expense of all other units in the system.

Finally, although the model achieves performance isolation, it unfortunately restricts the management of different types of resources to the same structure. As processes may use resources from their assigned SPU only, it is not possible to manage different types of resources in different groups. Like Opal's resource group abstraction, this restriction is inflexible.

4 A Generic Framework

Mungi's existing secondary storage management model varies cost with respect to utilisation (cf. Section 2). It discourages maximised disk space usage, which generally has little benefit to system performance anyway. Usage of other resources, however, needs to be maximised for increased system throughput and performance. This requires the specification of an amount of resource required for good performance.

The existing model, nonetheless, was not designed to specify an amount of resource to be allocated. Bank account balances could serve this purpose, e.g. by allocating more to a user with a high account balance. However, bank account balances vary with resource usage and salary deposition, and thus are not particularly well suited to this purpose. They are better suited to record a sense of usage history (as high usage will deplete an account). Furthermore, such a policy would significantly reduce the ability of users to control their resource usage according to their own requirements; effectively this would build too much policy into the kernel. Hence another abstraction is needed to specify an amount of resource to be used by an individual entity or a group of entities. This section presents the design of this abstraction with the requirements of Section 1 in mind.

4.1 Resource Consumption Rates

As Mungi's existing resource model is economic, an economic specification of resource needs is desired. To this end, *resource consumption rates*, a concept similar to rates in the Amoeba rental model, are introduced. These specify the maximum rate with which a particular resource is allowed to draw funds — how much of its income stream an entity is willing to commit to a particular resource. Logical entities (e.g., threads) are associated with a rate for each accountable type of system resource, and resource allocation is dependent on the rate specified and the *cost per resource unit*.

Together, rates and costs per unit resource provide much flexibility in policy construction. The system may utilise idle resources by lowering costs, while raising costs could be used to reserve resource units. Costs can serve as a reflection of node load in load balancing. As with backing store, setting arbitrarily high costs during high utilisation can be used to prevent maximised usage. A policy can also provide resource guarantees by committing to an upper cost limit.

As well, variable costs permit absolute or relative shares. The former, a quota based policy, would have a *fixed cost* per resource unit. The latter could have *zero pricing*, in which case usage is free as long as

there are idle units, and is in proportion to the specified rates when resources are in contention.

Finally, rates naturally supplement bank accounts. While accounts indicate usage history and bound the total resource usage of entities, rates determine the speed at which funds are consumed. The two abstractions give control over resource quantity and consumption time. For example, low resource rates and limited balances could be set for untrusted procedure calls. Users with little usage history could be permitted higher rates in accordance with system policy. Rates could also be adjusted to permit users to pay more for one resource and less for another resource while consuming funds at the same total rate.

4.2 Resource Groups

As mentioned earlier, resource isolation and prioritisation should be applicable to a group of entities (threads in this case). Any group abstraction must therefore fit in with the thread hierarchy used by Mungi (and most operating systems), with resource group membership orthogonal to the thread hierarchy for maximum flexibility. We call this abstraction *resource groups*, as in Opal where they were originally proposed [Chase et al., 1994].

Mungi threads belong to a resource group for each type of accountable system resource. Flexible thread grouping is possible since groups are unrelated to protection domains. Each group has a unique identifier, a *resource consumption rate*, resource usage data, creation and exit timestamps for accounting purposes, and a single bank account. The unique identifier distinguishes groups associated with the same account.

For finer control, resource groups are nested in a hierarchy per type of resource. This permits control over the total rate in a tree of groups. Protection of groups in this hierarchy is possible by validating access to group bank accounts during important group operations. Since validations are cached, this does not slow the operations, which are as follows:

1. If a thread spawns or moves a child into its own resource group, no access validation to its group's bank account is required.
2. When a thread spawns or moves a child into a newly created subordinate group, the thread requires a capability to its own group bank account. The new group is given a subset of its parent's rate and may be funded by another financial bank account.
3. If a child is started or moved into an existing group, directly or indirectly subordinate to its own, its parent must have a capability to the new group's bank account.
4. A thread may decrease the rates of groups directly or indirectly subordinate to its own if it has a capability to its own group account. This allows resource revocation from subordinate groups regardless of how these groups are funded. However, to increase a subordinate group's rate a capability to its bank account is needed.
5. When parent threads delete descendants, empty subordinate groups are removed and their rates added to their own group rates. Orphaned groups are adopted by groups above them in the hierarchy.

Figure 1 illustrates a resource group hierarchy with the Mungi thread hierarchy. Although this diagram has groups comprised of subtrees in the thread hierarchy, less structured groups are possible since threads can move descendants between groups.

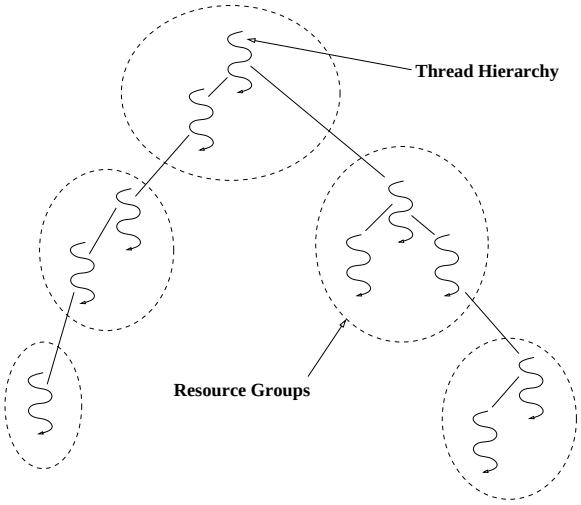

Figure 1: Resource groups and thread hierarchy

Resource groups also permit controlled resource usage during an invocation (via PDX) of an untrusted procedure. A thread may execute in existing or new resource groups during the call. When the call is completed, the calling thread resumes execution in its original resource groups. This logical change in groups during the PDX call is illustrated in Figure 2.

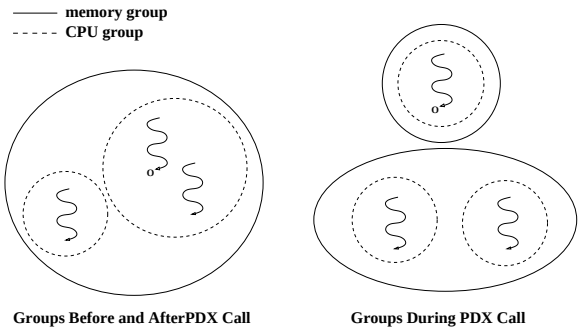

Figure 2: PDX call with caller thread `O` changing resource groups temporarily

Finally, it must be stressed that groups do not impose policy. Rather, user-level threads are responsible for defining resource allocation policies for descendants in subordinate groups. Unlike tickets and currencies, no resource redistribution policy is imposed when subordinate groups exit the system; parent groups only have their rates increased by the exiting groups' rates. It is completely up to the user-level threads responsible for the policy imposed on these groups to redistribute released resources among remaining subordinate groups if required. Similarly, to avoid imposing policy, the rates of groups are not increased to account for idle groups. Adjusting resource rights to give active entities the resources of idle entities is a policy imposed in Share to ensure fairness when sharing at an organisation level.

4.3 Incorporation into Mungi

Resource groups have little impact on the existing disk management model, simplifying their incorporation into Mungi. Bank accounts and operations like salary deposition and taxation are unaffected. Only increases in salary payments are needed to fund additional resource types.

On the surface, this increase in salary might seem

to affect secondary storage rent collection. Since object allocation is not limited by the resource group abstraction, users could feasibly spend all their funds on backing store, even parts of their increased salary intended for other resources. Other users could be denied their share of secondary storage. However, this is not really a problem. Disk is managed to permit the usage of idle storage and maximised usage is discouraged by raising storage costs as the resource grows scarce. As well, unlike other resources, buying more backing store does not really affect the performance of other users. Therefore, as long as a user has a financial bank account the purchase of more disk storage is legal. It may even be viewed as an advantage as users can choose to pay more for backing store without having to adjust resource consumption rates for secondary storage.

One other major concern is rent collection for resources like RAM. As with disk management, this is conducted by a user-level background thread to ensure kernel operations are efficient and that the Mungi kernel does not dictate resource management policy. The rent collector regularly traverses all groups in the system, deducting rent from their bank accounts. The rent charged is the product of the group resource usage, the present cost per unit resource, and the time of utilisation. Usually this is the time elapsed since the last rent collection, but may involve group creation and exit timestamps for groups that entered or left the system since the last collection.

Maintaining exit timestamps implies a delayed cleanup of resource groups. Empty groups are not immediately destroyed; the system retains group data until a thread with a capability to the OT, namely the rent collector, inspects it. This permits accurate accounting even if threads have short lifetimes and do not exist during rent collection. Since group data consists of a group's resource usage and creation and exit timestamps, the data retained by the system is minimal. Therefore, if rent collection occurs regularly, delaying the cleanup of group data has insignificant overhead on the system.

Finally, kernel support for resource group hierarchies is simple since group relationships mirror thread relationships. It merely involves maintaining pointers between group parents, siblings, and children.

5 Prioritising Physical Memory

While resource groups constitute a flexible management framework, mechanisms to prioritise resources and provide isolation for groups of threads are still needed to enforce resource consumption rates. This section considers the design of a mechanism for physical memory allocation. This resource can be multiplexed in both time and space, with the former achievable by controlling paging bandwidth. Paging bandwidth control is beyond the scope of this paper, and we will present the design of a mechanism only for prioritising physical memory frame allocation.

5.1 The Dilemma of Shared Frames

Although memory frames can be easily divided among entities, shared physical frames make fair accounting of memory resources difficult. As described in Section 3.7, one solution is to charge all shared resources to a single entity. (This is unfair, but acceptable if sharing is rare.) For Mungi, this is unacceptable since data sharing is encouraged by its single address space and shared memory is the basis for interprocess communication. Resource accounting could be greatly distorted if there is much sharing of resources between particular entities in the system (as it generally happens in other systems attempting to account for physical memory). For fairness, a mechanism is needed to solve this while charging frames only to entities using them.

5.2 A Fair Accounting Approach

[Chapin, 1997] states that physical memory control is mainly achieved by prioritising memory allocation and page replacement. This section introduces a fair and simple prioritisation scheme based on the clock page replacement algorithm. The clock algorithm is used since it (or variants of it) are used for page replacement in most operating systems. In particular, the existing Mungi virtual memory system employs a global clock algorithm with reference bit emulation.

To eliminate the dilemma of shared frames, this variant of the clock algorithm charges individual memory frames to only one memory resource group at a time. This provides a unique association between frames and groups, permitting straightforward usage accounting. A group is charged a frame if it sets the frame's reference bit (when one of its threads fault on the page, or when Mungi uses the page on its behalf). It remains charged for the frame until the page is replaced from physical memory, or the reference bit is set by a thread from another group. The latter case is referred to as a *page transfer*.

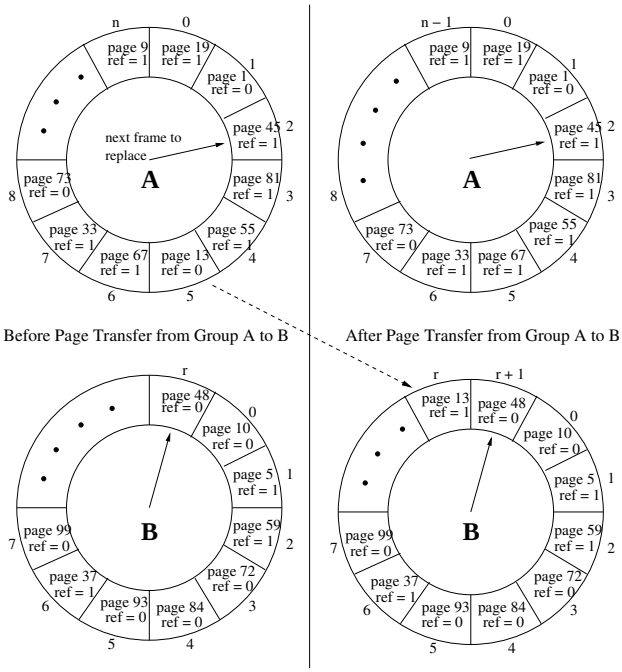

Figure 3: Page transfer between resource groups

Figure 3 shows a page transfer from group A to group B. Before the transfer, group A is charged for the frame marked page 13. This implies that sometime earlier group A had been responsible by setting the reference bit for page 13. Although the reference bit for page 13 has since then been cleared and the page unmapped from all threads by the page replacement algorithm, the frame is still charged to group A. Now when a thread from group B faults on page 13, it sets the frame's reference bit, resulting in a page transfer. As seen in the diagram, this causes page 13 to be charged to group B instead of group A.

Note from the example that page replacement is applied on a group basis, rather than globally. This is because global page replacement violates isolation. This approach does not imply localised page replacement; page faults from threads do not necessarily replace frames charged to their own groups. Rather a

frame from a group which is over its entitlement is selected for replacement.

One possible argument against this approach is its reliance on reference bits. A group with a large resource consumption rate might always keep its pages resident, allowing other groups to use them free of charge. This is not an issue since a group only pays for the frames it uses and its rate defines the maximum credit it is willing to pay. If the group's working set becomes too expensive, some pages will be automatically transfered to other groups using the same frames. Eventually this will lead to a fair distribution of costs. Even in the short term there is some statistical fairness, as a page will be charged to the first group using it.

5.3 Prioritised Memory Revocation

While the charging scheme described above is fair and simple, it still requires a method of selecting a resource group for page replacement. This method should only incur minimal overhead and must support a cost per frame. This cost is a function of memory utilisation, with an increasing value as free memory decreases. It must be controllable by the system administrator.

Minimum-funding revocation described in Section 3.6 seems to be a possible approach. Economically, this algorithm reallocates frames from entities paying less to entities paying more. To enforce a cost per frame, frames would be revoked until all entities are paying the appropriate amount per page. For efficiency, these revoked frames could be buffered until free memory is available.

Unfortunately, minimum-funding revocation is inefficient, with a tree data structure implementation requiring $O(log(n))$ operations, where n is the number of clients competing for memory. For a system where memory is prioritised only when there are no idle resources, this might be acceptable if page replacement is infrequent and accountable entities are few. However, this is unacceptable for our model since prioritisation depends on a cost per frame and revocation may occur even with available idle resources.

A set of group queues solves this inefficiency while also providing support for rates and a cost per frame. These queues are based on the array of 32 run queues employed in the 4.4BSD Operating System [McKusick et al., 1996] to manage processes. Whereas the 4.4BSD scheduler assigns a process to a queue according to its priority, this approach assigns a group to a queue according to its *bidding price*. This is defined as *the ratio of its resource consumption rate over the number of physical frames it is charged for*. It reflects the price the group is willing to pay per frame given its present allocation. A bidding price below the current frame price is an indication that the group holds too many frames – revoking some (assuming a fixed resource consumption rate) will increase its bidding price.

When scaled appropriately, the bidding price indexes into an array of 64 group queues, thereby locating the queue for the group.[1] As with 4.4BSD run queues, this array has an associated 64 bit vector identifying non-empty queues. Figure 4 illustrates this data structure.

A single ordered list would make the selection of the next victim group is fast, but all other operations would become expensive. By employing an array of queues, the expense of other operations is reduced, while the cost of selecting the next victim remains reasonably low (time to search the 64-bit vector).

[1] This scaling implies that the bidding price is treated as a discrete entity with 64 possible values.

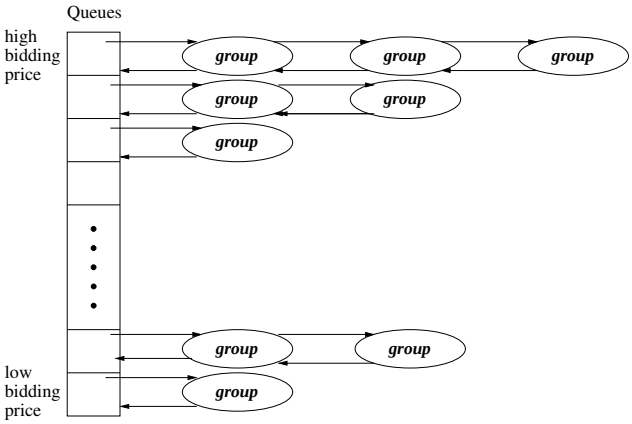

Figure 4: Group queueing structure for memory prioritisation

Given that the selection of a victim group is less frequent than the operations required to order groups, this poses no problems. With the array, fast selection of a victim group is possible by searching the bit vector. The lowest nonempty queue is indicated by finding the first bit set in the vector. Page replacement is subsequently possible with the first group on this queue. After page replacement this group is then appended to a queue reflecting its new bidding price.

Notice how choosing the first group trades the criteria of choosing the client with minimum funds for speed; fairness is still retained as groups are selected from the lowest nonempty queue. Another advantage of selecting the first group is that groups which have been in the queue longest are chosen first, thus protecting a group from losing too many frames too quickly.

5.4 Kernel Modification Details

Support for our fair charging scheme requires some substantial modifications to the existing Mungi virtual memory design. Kernel page fault handling now considers four cases of faults to correctly charge frames to groups:

1. If the faulting page is **not resident, but free memory is available**, I/O is initiated to load the page into a free frame. When the I/O is completed, the frame is mapped to the faulting thread and is charged to the faulter's group.

2. If the page is **not resident and free memory is unavailable**, the pager selects the first group from the lowest non-empty queue for page replacement. Appropriate I/O operations are initiated and, once these are completed, virtual memory mappings are established and the replaced frame is charged to the group of the faulting thread.

3. If the page faulted on is **resident and charged to a resource group**, the pager determines whether it may be mapped immediately without a page transfer. This occurs if the reference bit is set. If the reference bit is cleared but the faulter belongs to the same group as the frame, a mapping is likewise immediately returned. Otherwise, a frame transfer is needed to charge the frame to the faulter's group before mapping the page to the faulter.

4. Otherwise the page faulted on is **resident and not charged to a resource group**. Pages of this type are buffered on a dirty list. A fault on

a buffered dirty page charges the frame to the faulter's group.

Note that although buffering improves system performance, it introduces two complications. Firstly, uncontrolled buffering of dirty pages leads to high cost when revoking dirty pages which require writing to disk. Secondly, buffering dirty pages does not permit policy to reserve frames by raising the cost per frame. These issues are a part of controlling paging bandwidth and are subject to future research.

In all cases where no free memory is available, page replacement may be required for the faulting group prior to charging the frame to its new group. If the frame's resource group is exceeding its rate, the pager invokes the clock algorithm to select a replacement frame. Other frames are unmapped and have their reference bits cleared. The replaced frame may then be buffered in memory and dirty frames cleaned when necessary.

Also, notice how this algorithm only replaces one page per fault for resource groups unable to pay for frames. It serves to limit the frames of groups over their rates while permitting fast fault handling. This is sufficient if memory only needs to be prioritised on page replacement (which is the case when a zero cost per frame is specified). An asynchronous thread, such as the page cleaner, must therefore periodically revoke frames when group rates are reduced or the cost per frame increased. The algorithm to achieve this is as follows:

1. The cleaner is initially awaken by one of three events: an increase in frame price, a decrease in a resource group's rates, or a timeout indicating a periodic cleaning sweep is due. The number of frames replaced each round is limited to minimise the processing time spent and the amount of disk traffic produced by the cleaner.

2. Firstly, the cleaner locates the lowest nonempty queue. If the lower limit of the bidding prices represented by this queue exceeds the frame price, no groups require revocation (all pay at least the present price for what they consume) and the thread can sleep again.

3. Otherwise, if the queue covers a range less than the cost per frame, the first of its groups is selected. Some of its frames are replaced before appending the group to a queue reflecting its new bidding price. This is repeated while the lowest queue is not empty and the target number of replaced frames is not reached. Should the current queue become empty, the cleaner must locate the next lowest nonempty queue.

4. As groups move up the array with revocation, the queue specified by the cost per frame will eventually be the lowest non-empty queue. For this queue, all groups paying less than this cost require revocation. The thread sleeps when no more groups require revocation or the maximum number of frames to replace each period is reached. If the latter occurs, a timeout is set to wake the cleaner to repeat the process later.

Finally, note that in both algorithms frequent movement of groups between queues do not occur unless the cost per frame or group rates change often. Group memory usage generally reaches a maximum determined by the group rate or the working sets of group threads. Little overhead is therefore incurred in maintaining queues; the movement of groups between queues usually settles down quickly.

6 Experimental Results

Mungi is built on top of the L4 μ-kernel. Current versions execute as user-level tasks on top of the MIPS [Elphinstone et al., 1997] and Alpha [Potts et al., 2001] versions of L4. Our prototype model was implemented with Mungi version 1.2 using L4/MIPS. Relevant machine hardware characteristics are a 100MHz R4600 CPU, 2-way set associative 16KB instruction and data caches with 32 byte line sizes, and a main memory size of 64MB.

Mungi's VM paging system is undergoing major revision, as the existing implementation is a stop-gap version unsuitable for benchmarking. Consequently we did not benchmark any actual paging activity, but only tested the control of resident set sizes, and migration of frames between resource groups.

A simple experiment was conducted to see whether rates were enforced by our prototype model with Mungi on L4/MIPS. In this experiment a thread spawns two threads in its own protection domain and sets the cost per frame to 1024 units. (Such a thread could be a trusted "system" thread, or a "user" thread which manages sub-threads within its own share of the system.) The child threads are assigned to separate memory resource groups each entitled to 50 frames. These threads dirty pages in an object of 100 pages for 50 seconds. Child *thread1* continuously loops and dirties the first 75 contiguous pages, while *thread2* loops and dirties the last 75 contiguous pages. In this manner, the threads share 50 frames between them.

During the test two parameters are changed. The rate of *group2*, the group for *thread2*, is halved 20 seconds into the test. The cost per frame is set to zero 10 seconds later. The results are illustrated in Figure 5, a graph of the number of frames charged to each group per page fault incurred.

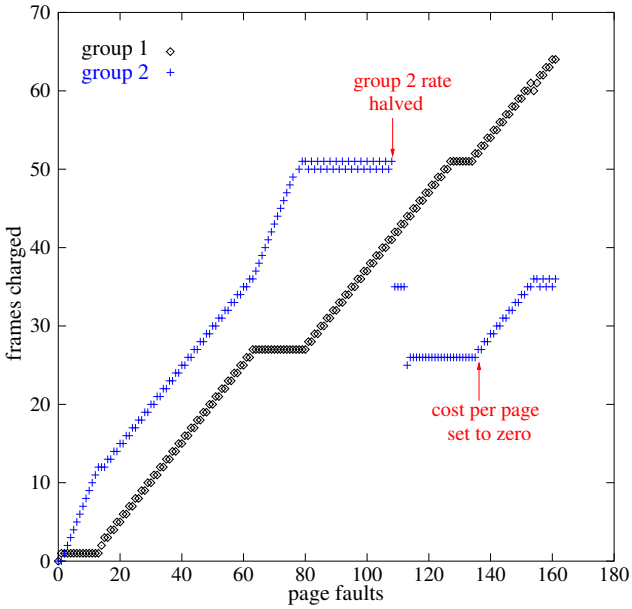

Figure 5: Experimental results

Initially, each child thread sleeps and periodically awakes to check a flag signalling it to start dirtying pages. Child *thread2* sees the flag set and dirties pages first. A few faults later, *thread1* likewise begins to dirty pages. It is charged frames for the first 25 pages of the object, but ceases to incur new charges while it dirties frames already resident and charged to *group2*. This is illustrated in the plateau between 60 to 80 faults. New charges to *group1* only occur

after *group2* reaches its maximum of 50 frames and is subject to page replacement.

At 20 seconds, Figure 5 shows a sharp drop in the frames charged to *group2* when the rate for *group2* is halved and the background cleaner immediately replaces some of its frames. The cleaner then sleeps for a second before continuing to replace *group2* frames until *group2* has roughly 25 frames remaining. No more frames are replaced by the cleaner as both groups are executing within their rates.

Finally, when zero page pricing is set at 30 seconds, both groups steadily incur new frame charges. Several page transfers occur during these last faults, causing the rather bumpy increase for *group1*. Page faults cease when *group1* has 64 frames and *group2* has 36 frames, a reflection of the ratios of the threads' resource consumption rates.

Overall, these experimental results are encouraging, showing how shared frames can be fairly charged among different entities. It demonstrates how frames are successfully allocated according to group rates and the cost per frame.

7 Conclusion

Operating systems require a generic framework to manage a diverse range of resources, as well as mechanisms to prioritise resource allocation and isolate processes from resource contention. In Section 1 we specified the requirements of a generic resource management framework. To satisfy these requirements, a new kernel abstraction, the resource group, was introduced to support allocating proportions of resources to groups of threads. The kernel was augmented by data structures for supporting physical-memory priortisation, and by mechanisms which allow resource groups to control this priorisation. The design meets the following goals:

Fairness: Resource isolation and prioritisation were stressed as necessities for fairness in resource management. The mechanism designed met these requirements for physical memory. In particular, it provides fair accounting of shared frames, a problem often overlooked.

A fair resource framework should also permit policy to consider usage history when allocating resources. Our framework achieves this using bank accounts from the existing storage management model. Bank accounts serve as a means of tracking usage history; high and low resource usage is indicated by low and high account funds respectively.

Flexibility: A resource group is a flexible and simple abstraction on which more complex policies can be built. Threads are assigned to groups for each type of accountable resource in the system and flexible grouping is possible since groups are unrelated to protection domains.

Moreover, the kernel does not redistribute the resource rights released by exiting resource groups among other groups, nor does it increase the rights of groups with the resources of idle groups. In this manner, no policy is imposed by the kernel. Rather, the system enforces policy defined at user-level: threads in higher groups define the policy to impose and determine the resources allocated to subordinate groups. This permits policies where entities choose to pay more or less for resources within policy constraints. Resource groups also permit absolute or relative proportions of resources to be specified.

Performance: The existing storage management model has minimal accounting overhead since most accounting is conducted outside the kernel. This approach is employed for accounting resources allocated to resource groups, ensuring kernel operations remain fast.

Performance has likewise been addressed in the design of a queueing structure for the quick prioritisation of frame allocations. This approach is faster than algorithms such as lottery scheduling and minimum-revocation funding.

Protection: Protection for our framework has been provided by the validation of group bank accounts in group operations. These operations ensure threads do not violate policy constraints by illegally modifying the rates of groups in the group hierarchy.

Simplicity: Our physical memory prioritisation relies on a simple concept of charging. The unique association of frames to groups ensures users are charged only for the frames they use without complicated division of costs for shared frames.

Overall, these achievements are an important step towards a generic resource management framework for Mungi. The results of our simple experiment have been encouraging, demonstrating the ease at which our resource group abstraction can be applied to physical memory. This group abstraction fits in well with Mungi's other fundamental abstractions, particularly with the Mungi thread hierarchy. Since these fundamental abstractions are not specific to Mungi, our framework can be easily adapted to other systems too.

Further development is needed, nonetheless, especially if resource isolation for physical memory is to be fully achieved. This requires an implementation of paging bandwidth control to prevent thrashing threads from degrading the performance of others and thus complete the isolation property. Actual performance measurements with benchmarks and real workloads would then give more evidence of the strengths of our design.

References

[Anderson et al., 1986] Anderson, M., Pose, R., and Wallace, C. S. (1986). A password-capability system. *The Computer Journal*, 29:1–8.

[Chapin, 1997] Chapin, J. (1997). A fresh look at memory hierachy management. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 130–134, Cape Cod, MA, USA.

[Chase et al., 1994] Chase, J. S., Levy, H. M., Feeley, M. J., and Lazowska, E. D. (1994). Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12:271–307.

[Drexler and Miller, 1988] Drexler, K. E. and Miller, M. S. (1988). Incentive engineering for computational resource management. In Huberman, B. A., editor, *The Ecology of Computation*, Studies in Computer Science and Artifical Intelligence, pages 231–266. North-Holland, Amsterdam.

[Elphinstone et al., 1997] Elphinstone, K., Heiser, G., and Liedtke, J. (1997). *L4 Reference Manual: MIPS R4x00*. School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia. UNSW-CSE-TR-9709. Latest version available from `http://www.cse.unsw.edu.au/~disy/L4/`.

[Heiser et al., 1998a] Heiser, G., Elphinstone, K., Vochteloo, J., Russell, S., and Liedtke, J. (1998a). The Mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928.

[Heiser et al., 1998b] Heiser, G., Lam, F., and Russell, S. (1998b). Resource management in the Mungi single-address-space operating system. In *Proceedings of the 21st Australasian Computer Science Conference (ACSC)*, pages 417–428, Perth, Australia. Springer-Verlag. Also available as UNSW-CSE-TR-9705 from `http://www.cse.unsw.edu.au/school/research/tr.html`.

[Kay and Lauder, 1988] Kay, J. and Lauder, P. (1988). A fair share scheduler. *Communications of the ACM*, 31(1):44–55.

[Krueger and Chawla, 1991] Krueger, P. and Chawla, R. (1991). The Stealth distributed scheduler. In *Proceedings of the 11th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 336–343, Los Alamitos, CA, USA.

[Larmouth, 1975] Larmouth, J. (1975). Scheduling for a share of the machine. *Software: Practice and Experience*, 5:29–49.

[McKusick et al., 1996] McKusick, M. K., Bostic, K., Karels, M. J., and Quarterman, J. S. (1996). *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley.

[Mullender and Tanenbaum, 1986] Mullender, S. J. and Tanenbaum, A. S. (1986). The design of a capability-based distributed operating system. *The Computer Journal*, 29:289–299.

[Potts et al., 2001] Potts, D., Winwood, S., and Heiser, G. (2001). *L4 Reference Manual: Alpha 21x64*. University of NSW, Sydney 2052, Australia. UNSW-CSE-TR-0104. Latest version available from `http://www.cse.unsw.edu.au/~disy/L4/`.

[Sullivan and Seltzer, 2000] Sullivan, D. and Seltzer, M. (2000). Isolation with flexibility: A resource management framework for central servers. In *Proceedings of the 2000 USENIX Technical Conference*, pages 337–350, San Diego, CA, USA.

[Tanenbaum et al., 1986] Tanenbaum, A. S., Mullender, S. J., and van Renesse, R. (1986). Using sparse capabilities in a distributed operating system. In *Proceedings of the 6th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 558–563. IEEE.

[Thompson and Ritchie, 1974] Thompson, K. and Ritchie, D. M. (1974). The UNIX time-sharing system. *Communications of the ACM*, 17:365–375.

[Verghese et al., 1998] Verghese, B., Gupta, A., and Rosenblum, M. (1998). Performance isolation: Sharing and isolation in shared-memory multiprocessors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 181–192, San Jose, Ca, USA.

[Vochteloo et al., 1996] Vochteloo, J., Elphinstone, K., Russell, S., and Heiser, G. (1996). Protection domain extensions in Mungi. In *Proceedings of the 5th IEEE International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 161–165, Seattle, WA, USA.

[Waldspurger, 1995] Waldspurger, C. A. (1995). *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Dept of EECS, MIT.

[Waldspurger and Weihl, 1994] Waldspurger, C. A. and Weihl, W. E. (1994). Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–11, Monterey, CA, USA.