Resource Management in the Mungi Single-Address-Space Operating System

Gernot Heiser, Fondy Lam and Stephen Russell

Department of Computer Systems
School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia
G.Heiser@unsw.edu.au, http://www.cse.unsw.edu.au/~ disy

Abstract. We present the accounting system used for backing store management in the Mungi single-address-space operating system. The model is designed such that all accounting can be done asynchronously to operations on storage objects, and hence without slowing down such operations. It is based on bank accounts from which rent is collected for the storage occupied by objects. Rent automatically increases as available storage runs low, forcing users to release unneeded storage. Bank accounts receive regular income. A taxation system is used to prevent excessive buildup of funds on underutilised accounts.
The accounting system is mostly implemented at user level, with minimal support from the kernel. As a consequence, the accounting model can be changed without modifying the Mungi kernel.

1 Introduction

Fair allocation of shared resources is a traditional duty of an operating system. The most important resources are CPU time, RAM and secondary storage. The CPU is a pre-emptible resource, users of which are *scheduled* for short periods of time, based on considerations such as total system throughput and process priorities. RAM is usually also treated as a pre-emptible resource and assignment to users is based on similar considerations as in the case of CPU time. Secondary storage, by contrast, is an example of an *allocated* resource: As it is used for long-term data storage it is generally not pre-emptible. The amount of secondary storage allocated for a particular purpose has little impact on system performance, but much on whether or not users can get the service they want from the system. The system must ensure that individual users cannot starve others by using an unreasonable amount of storage.

The major challenge for the system is to control the proliferation of unneeded objects in a system in which it is difficult to use conventional garbage collection methods. This paper examines the problem of managing the allocation of secondary storage. It does so in the specific context of the Mungi operating system developed at UNSW.

2 Overview of Mungi

Mungi [Heiser et al., 1994] is a single-address-space operating system (SASOS); i.e., all processes throughout all computing nodes in the system share the same virtual address space. That address space contains all data, transient as well as persistent. Within this *single-level store* data are identified through their (64-bit) addresses.

Virtual memory is allocated in contiguous, page-aligned segments called *objects*, which are also the unit of protection: A process is granted certain access rights to all or none of an object. Access is controlled via password capabilities: When an object is created, the system returns a capability to the user, which contains the object's base address and a password. Such a capability grants full (read, write, execute and destroy) rights to the object and is called an *owner capability*. A process holding an owner capability (an "owner" for short) can register less powerful capabilities for an object.

Capabilities can freely be stored or passed around without system intervention. They are protected from forgery by their password, which is registered in a global, distributed data structure called the *object table* (OT). When validating a capability the system compares its password with the list of valid passwords stored in the OT and grants access if the requested operation is compatible with the access mode stored with the password in the OT.

Backing store for virtual memory objects is allocated on demand, i.e., no backing store is reserved when an object is first created. Once a particular page of an object is first accessed, the system allocates backing store for that page. Virtual memory is cheap in this system (there are 16 billion gigabytes of it), but allocated pages are, of course, limited by the amount of disk connected to the system. Hence the use of backing store needs to be controlled. At first glance, it might seem that automatic garbage collection would solve the problem. However, as explained in Section 3.1, we are not addressing the same issues as those faced by garbage collection. The objective of our system is to discourage users from over-using secondary storage.

3 Previous approaches

The simplest approach to secondary storage management is to ignore the fact that it is a limited resource, and allocate storage to users on demand. This may lead to all storage being exhausted, in which case a process attempting to allocate data will fail, and users are asked (by the system operator) to "clean up".

This approach is reasonable on a single-user system or on a system shared by a number of cooperating users, such as a work-group environment. It is unsuitable to a general multi-user environment. The ability of individuals, by malice or accident, to prevent others from using the system productively is unacceptable in general.

3.1 Automatic garbage collection

Data which have no more use in the system constitute *garbage*, which should be removed to minimise waste of resources. Garbage collection on secondary storage is usually integrated with the file system: Reference counts (e.g. the number of links to a file) are used to detect and remove inaccessible data.

This is, however, insufficient for controlling disk space usage. A file which contains nothing of interest to any user, and which will therefore never be accessed again, is not "garbage" in the above sense, as it is still referenced by a directory entry and it is therefore impossible for the system to discover its uselessness and remove it. In the end, users must decide whether they want to keep or remove data. The best the system can do is to provide an incentive for users to clean up (or a penalty for failing to do so).

3.2 Quotas

The usual approach is to introduce fixed, per-agent storage limits or *quotas*. These are usually set on a per-user basis, where users are allowed to allocate storage up to their quota, after which point no further allocation is possible (i.e., the "penalty" for not cleaning up is to prevent users from storing further data). Users are fully responsible for managing their own storage within their quota. Such a system is implemented in most commercial multi-user systems such as UNIX.

One of the main drawbacks of quota systems is their inflexibility. Users may exhaust their quota, and thus be forced to free up storage, even though there is plenty of free storage available system-wide, and even though a user may need a large amount of storage for only a very short time. This approach cannot make any difference between more or less valuable data, or distinguish between long-term or short-term storage.

The inflexibility of quota systems becomes evident when looking at the resource use they permit. Quotas may be set such that the sum of all quotas does not exceed the available storage. In this case the system cannot run out of storage, but as many users will not exhaust their quota, a large fraction of the resource will always be unused. Alternatively, the resource can be *over-committed* (by allowing the sum of all quotas to exceed the total amount of available resource). In that case, the system may not be able to grant a request for resources even though the requester is well within her quota.

Quotas can have obscure effects. For example, consider a UNIX file system with quotas. Every file has a user dedicated as its *owner*, and every user has a disk quota. Disk space allocated to files owned by a particular user is charged against that user's quota. Consider a malicious user A who (hard-) links all accessible files of another user B which reside on a file system on which A has a directory. If A creates these links in a directory which is not accessible to B, B cannot discover these alternative links to her files — she can only infer their existence from the link count on her files. B can now no longer remove these files (she can only unlink her own paths) and therefore they keep being charged against her quota. By using this method systematically, A can mount a denial-of-service attack against B, because B will eventually run out of disk quota. B cannot even detect who is attacking her. The only way out is for B to ask the superuser to run a search over all files in the respective partition.[1]

A final, very significant, drawback of quotas is that they need to be checked and updated whenever resource usage changes, e.g. every time a file is created, destroyed, extended or truncated. A system-wide directory of all user quotas could be kept which must be updated consistently whenever any file changes size, a huge problem in a distributed system. The problem can be eased by keeping per-partition quotas, at the expense of making the system less transparent to users, who now need to be aware of a *set* of quotas, one for each file system partition.

3.3 Economic models

An alternative approach is based on an economic view of resources. While in a quota system resource use is essentially free, but limited, an economic model puts a cost on

[1] Note that this problem cannot be solved by garbage collection: The files are not garbage, as they are linked to A's directory.

any resource use, and leaves it to users (or "clients") to decide whether the benefit obtainable from using a resource outweighs the cost. While computer operators have been charging for machine time since the early days of computing, and on many commercial timesharing system it has been possible to buy scheduling priority with "real" dollars, only few attempts have been made in the past to use an economic approach for internal resource allocation in a context where "real" money is not necessarily an issue.

"Money" and "rent" in the Monash Password Capability System Possibly the earliest use of an economic model for resource management was in the Monash *Password Capability System* [Anderson et al., 1986]. That system charges size-dependent *rent* for each object, and defines an object as garbage when it cannot pay its rent. To this end, every object is associated with some amount of *money*, where money is defined as a transferable right to use system services. This association is done by assigning a monetary value to every capability. Every object has a unique *master capability*, from which (potentially lesser) capabilities can be derived. Monetary values belonging to derived capabilities are subsets of the respective master capability's money, and withdrawal from a derived capability's money implies a withdrawal from the master. A *rent collector* periodically scans the address space and withdraws money from the master capabilities of all objects. Garbage objects (who cannot pay their rent) are removed.

This system avoids the problems associated with garbage collection by shifting the responsibility for resource management to users. A user who wants to be certain that a particular object continues to exist must ensure that sufficient money remains associated with the object's master capability. Processes (being objects themselves) can do this by transferring some of their own money to other objects. Besides freeing the system from the need to decide what is garbage, this system has the significant advantage that all accounting is done "off-line" (by a background process) so, unlike a quota system, user operations on objects are not slowed down by the need to perform accounting.

Amoeba's "bank accounts" Amoeba's resource management system [Mullender and Tanenbaum, 1986] is also based on the idea that money is used to pay for resources. Amoeba achieves this by introducing *bank accounts* as objects in their own right, rather than associating a monetary value with each object. Accounts are maintained by a *bank account server*, which is contacted for transactions on accounts.

Bank accounts are typically associated with individual human users of the system, whom they provide with the means to purchase services. This approach has the advantage that most users normally only need to be concerned with a simple account, which makes the accounting system quite transparent. Another advantage is that different subsystems can implement their own accounting policies (and one can easily envision setting up "competing" servers providing similar services with different accounting policies, leaving to users the decision which policy is best suited to their needs). This is supported by the provision of several, possibly convertible, currencies.

The system is flexible enough to allow implementation of a quota system (where disk blocks are "purchased" on allocation, and money is refunded on deallocation) or a rent system, where storage usage implies a continuous flow of money from client to server. The latter requires either the client to pay the server a sufficient amount in

advance, or regular communication between client and server (e.g. via an *alert* mechanism) to keep the money flowing. The client needs to *trust* the server to provide as much of a resource as the client has paid for.

Drexler and Miller's market model [Drexler and Miller, 1988] present a detailed discussion of various "ecological" models for resource management. Their models of storage management envision storage providers optimised to maximise storage usage, by adjusting the price of storage to market demand, determined by a bidding process. More important objects would bid more for space than less important ones. Objects recover their rent by charging their clients for use. Objects which run out of funds are garbage collected. Alerts can be used by cash-stripped objects to ensure their survival.

The model carries significant overhead, which restricts its use to fairly heavyweight objects. Also it would seem that, in order to make full use of what it offers, applications would be forced to constantly "play the markets" for the best deal, or lose out. For example, cache sizes must constantly be adjusted according to the price of storage as otherwise performance or account balance would suffer. We suspect that this model would create an environment which is rather different from the "look-and-feel" of present computing systems. This is not necessarily bad, and it would be interesting to see such a system in action. However, introducing such a dramatically different resource management model at the same time with another change in paradigm (i.e., the single address space) would make fair performance evaluations of the system more difficult and, most likely, also increase the reluctance to migrate to the new system.

However, the main drawback for our purposes is that the system does not operate off-line, but requires accounting operations whenever storage is allocated or freed.

Opal's resource groups The Opal SASOS [Chase et al., 1994] uses a form of bank accounts called *resource groups*. Objects are created with a reference to a resource group, which is charged for the resource use. This is combined with reference counting based on entities explicitly registering "an interest" in a particular object; objects are deleted when that reference count becomes zero.

Min-funding revocation [Waldspurger and Weihl, 1996] present the *min-funding revocation* algorithm, where agents hold *tickets* which represent *relative rights* to resources, that is the amount of a resource a particular ticket can buy depends on the value of the ticket as well as on the total value of tickets which are bidding for the particular resource. When a resource is released (voluntarily, or as a result of being out-bid) the respective ticket value is released to the holder. This system revokes resources with little notice and seems more appropriate for allocating caches or RAM than backing store. Again, accounting is done on-line, which to us negates one of the main advantages of economic accounting models.

4 Mungi's bank accounts

Many of the above economic schemes are based on the assumption that users can perform their tasks more effectively if they increase their resource usage, as long as "the

price is right", and that the system should encourage high average resource usage. This is clearly the case for physical memory or other cache stores, where system throughput is expected to be increased if all such storage is used, and it is the task of the accounting system to ensure that different users get a fair share of the resource (and the resulting performance advantage).

However, it is not clear that it is generally advantageous for the system to maximise utilisation of backing store. Users have an inherent need for some amount of storage, dictated by the amount of data involved in their operations. On top of this, garbage tends to accumulate, due to faulty programs or because some information loses its value over time. We believe that only the users themselves can make the decision whether a particular object is still needed — the actual removal of garbage objects must be fully under user control rather than being done automatically by the system.

Furthermore we want a system which is easy to use, and which is similar enough to established approaches that the difference is not evident to most applications. In particular we do not want to require users to maintain explicit reference to persistent objects, as these tend to be unreliable (due to cycles, or users "forgetting" references).

4.1 Basic model

Our model [Lam, 1995] is also based on bank accounts. Whenever an object is created, a reference to a bank account, which is to pay for the object's backing store, is supplied to the system and recorded in the OT. As the main intention is to charge for backing store (not virtual memory) and a newly allocated object is not using any backing store, no accounting is required at that stage. The system only confirms that a valid bank account has been supplied. This is like validating access to any other object; validation information is cached and hence validations are very fast [Vochteloo et al., 1996; Heiser et al., 1998]. Object creation (or destruction) is therefore not slowed down significantly by accounting. Furthermore, no accounting operations are performed when backing store is allocated to, or removed from, an object.

As in the Password Capability System (see Section 3.3) the actual accounting is performed by a background process, the *rent collector*. That process traverses the OT and for each object charges the associated bank account for the actual amount of backing store allocated to it. If the rent collector finds an account overdrawn, this account is marked as such and can no longer be used to create new objects.[2] If it finds an invalid bank account reference (due to the bank account having been destroyed, or converted into a "regular" object) the corresponding object is removed.

To make up for the constant drain on accounts, a source of income is required. We therefore associate a *salary* with each account. A *pay master* process periodically visits all accounts and deposits an appropriate amount.

[2] As access validations are cached, this does not necessarily take immediate effect. However, validation caches are guaranteed to be flushed regularly, hence the revocation of rights, including the right to use a bank account to create objects, can only be delayed for a certain maximum amount of time.

4.2 Storage cost

One of the main advantages of economic models is their ability to respond (by increasing the price) to situations where demand for a resource exceeds supply. In our model this means that the rent charged to objects increases as free storage becomes scarce.

In order to prevent the system from ever completely running out of free storage, we can impose a rent which approaches infinity as free storage approaches zero. That way, when storage becomes scarce, rent increases to arbitrarily high levels, forcing users to free up storage (as they can no longer afford to pay) and thereby increasing availability.

A main goal of our model is to be as transparent to users as possible. Hence we want to avoid drastically varying storage costs (which would require users to make frequent decisions on which objects to keep and which to remove) as much as possible, and thus restrict significant variations of the basic rent to the case of a heavily utilised system. This implies using a rent formula which is almost constant at low utilisation, increases slightly with increasing utilisation, and only raises sharply with very high utilisation.

A "smooth" (i.e., analytical) cost function with zero derivative for an empty system is preferable. We use for the cost ρ per unit of storage (e.g., a page):

$$\rho(\xi) = 1 + 4p\xi^2 \exp\left(\frac{\xi}{1-\xi} - 1\right), \tag{1}$$

where ξ ($0 \leq \xi \leq 1$) is the storage utilisation of the system ($\xi = 0$: empty, $\xi = 1$: full), and p is a parameter which the system administrator can use to determine how quickly the storage cost increases with increasing system utilisation. That parameter has an intuitive meaning which becomes clear when we observe that $\rho(0.5) = 1 + p$, i.e., p is the relative increase in rent of a half saturated over an empty system. Fig. 1 shows the cost function for various values of p.

4.3 Income and taxation

A fixed income has the disadvantage that an account which is used very little can accumulate large amounts of money. Consider a user who goes on an extended leave after some thorough "cleaning up" (removing most of his objects). Income during the period of absence would far exceed rent charged to his account, with the effect that the user would be very "rich" upon his return. He could then (for a short time, at least) allocate vast amounts of storage, effectively "buying out" other users. This is not the intention of the system, as incomes are supposed to reflect the share of the system a user can get.

To prevent such distortions we introduce a form of taxation.[3] Our taxation formula should have virtually no effect for accounts with a low balance (relative to income), while limiting the funds that can be accumulated. If we define β to be the account balance divided by salary ($0 \leq \beta \leq \infty$), we require an analytic function which approaches identity (i.e., no taxation) at $\beta = 0$ and has vanishing slope (i.e., saturates) at $\beta = \infty$. In addition we require the function to be strictly monotonic. We use the tax function

$$\tau(\beta) = b\left[1 - \exp\left(-\frac{\beta}{b}\right)\right], \tag{2}$$

[3] This is different from real economies, where taxation is imposed to provide the government with income.

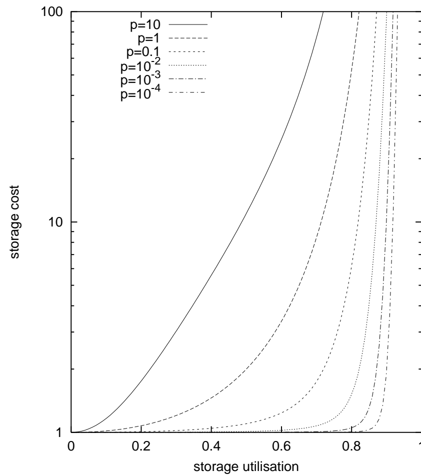

Fig. 1. Storage cost, ρ, as a function of storage utilisation, ξ, for various values of the parameter p.

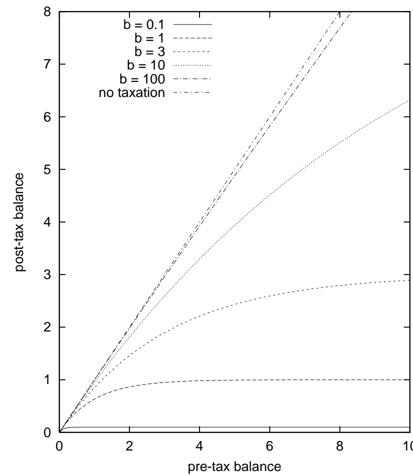

Fig. 2. Effect of taxation for various values of the balance limit b.

where b is again a configuration parameter. We observe that $\tau(\infty) = b$, giving b the meaning of the maximum multiple of income which can accumulate in any account (post tax). Fig. 2 shows the effect of taxation for different values of the parameter b.

4.4 Timing issues

Rent collection should happen reasonably frequently to prevent distortions from arising. Once a day (during times of low system usage) seems to be an appropriate frequency; the current charge per page is determined once according to Eq. 1 at the beginning of the rent run. The exact timing of rent collection is uncritical, as a time stamp is kept in each object's OT entry specifying when rent was last collected. The rent collector charges for the actual time elapsed since the time stamp, i.e., the rent deducted is the product of the present rate and the actual time elapsed.

The pay master is also run at roughly regular intervals. It uses a time stamp in the bank account to scale the basic salary by the time elapsed since the last deposit.

Taxation happens immediately before salary deposition. This can potentially cause problems when salary is not paid at regular intervals, as accounts where salary is deposited more frequently (and in smaller increments) would be excessively taxed if Eq. 2 was applied naively. However, taxation can also be scaled by the elapsed time interval, by replacing b in Eq. 2 by $b/\Delta t$, where Δt is the time elapsed since the last salary was deposited. This cannot completely prevent that results depend on the frequency of taxation. However, the differences are quite small even when comparing once-a-day deposits with several daily runs.

For the normal case of daily runs with slight variations of timing (of the order of a few hours) the difference will be hardly noticeable. This is shown in Fig. 3, where,

starting from an initial balance of zero, for ten successive days the remaining balance is taxed, new salary deposited, and a constant rent is charged. This is done either in exact daily intervals, or in intervals of mean length of one day and a standard deviation of one hour; the times add up to exactly ten days in either case. The resulting difference in the final balance is only a few percent.

Short-lived objects are charged only if they happen to exist during the rent run. If rent runs are scheduled at fixed times, users could conceivably adapt their activities to minimise storage usage during that time. If they are able to do this, then they obviously do not really **need** that storage, but only make use of an idle resource (e.g. to increase performance). From the system management point of view this is not objectionable. If, however, this was conceived as a problem, it could be countered by performing the rent run at random times, possibly several times a day. Rent collection does not need to be coupled to salary deposition and taxation, hence there is no reason why rent could not be collected several times a day, while salaries are only deposited once a day.

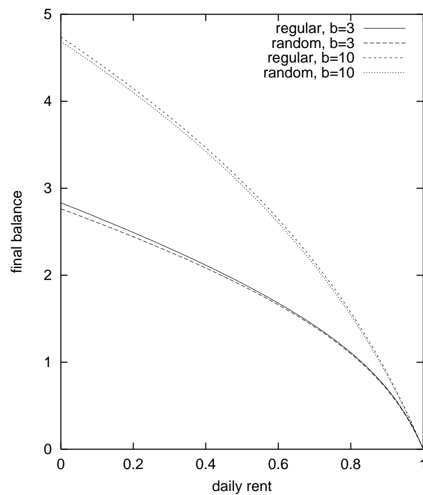

Fig. 3. Effect of regular vs. random taxation intervals, assuming constant rent charges.

```
typedef struct {
   address   object;
   int       size;
} BA_charge;

typedef struct {
   Cap_t     parent;
   float     salary;
   float     balance;
   time_t    last_deposit;
   float     alert_thresh;
   void      (*alert)();
   float     total_charge;
   int       n_charges;
   BA_charge charges[];
} Bank_account;
```

Fig. 4. Bank account data structure.

Alternatively it would be possible to introduce, in addition to the rent, a once-per-lifetime object charge. If this is collected *at object deletion time,* it does not slow down object operations as deletions can be performed asynchronously. That approach, however, requires some (simple) change to the kernel support presented in Section 5.1.

5 Bank account objects

5.1 Kernel support for accounting

The kernel support for the accounting system is minimal, as almost everything can be done at user level. The kernel, while not having to know about the operation of the accounting system, only needs to ensure that users cannot bypass it — an example of the separation of policy and mechanisms in Mungi.

Kernel support consists of two parts: The kernel ensures that

1. every object, when created, has a valid bank account, and
2. bank accounts cannot be tampered with.

The first point requires that a bank account reference (read capability) is passed to the ObjCreate system call, and that the kernel verifies that this capability actually refers to a financial bank account. To this end, the OT entry for each object contains a flag is_acct to identify bank accounts, plus a flag is_financial only relevant to bank accounts.[4] On object creation the kernel verifies that both flags are set in the object referred to by the bank account capability provided by the user. That capability is then recorded in the new object's OT entry.

To make bank accounts tamper-proof the kernel only allows creation of bank accounts by processes which have read access to the OT. (Note that reading the OT gives access to all passwords, and hence to all objects. Read access to the OT is like having an effective UID of "root" in UNIX.)

5.2 Bank account operations

Fig. 4 shows the bank account data structure. The charges array contains the list of objects which were charged to the account during the last rent collection cycle, together with the amount of backing store consumed by each. The field total_charge is the total amount which was charged by the last run of the rent collector. This is equal to the sum of all sizes in the charges list, times the present storage cost (Eq. 1). The alert procedure is called when the account balance falls below alert_thresh to notify the owner.[5]

Bank accounts receive their income from parent accounts. There is a special "root" account with infinite income from which all first-order accounts receive their income. The paymaster on each run first updates the timestamp of the root account, and then traverses the list of other accounts. For each account it taxes the present balance, debits the parent for the salary earned (after first processing the parent if it has not already received its salary; this can be checked by comparing the timestamp with the root), updates the balance with the salary and updates the timestamp.

Bank accounts, being objects themselves, also pay rent for their storage (which is usually charged to their parent account). Bank accounts are created by the *bank manager* upon request from a user process (which might be the system administrator). The

[4] Both flags are part of the access information kept in the validation cache.

[5] This call uses PDX, Mungi's protected procedure call mechanism [Vochteloo et al., 1996], to execute in the owner's protection context.

request must be accompanied by a read capability to a parent account from which the new account is to draw its salary. The system administrator holds a read capability to the (infinitely rich) root account and can thus create new income streams. The bank manager returns to the caller a bank account read capability, and keeps all owner (or write) capabilities to itself. This ensures that modifications of bank accounts can only be done by the accounting software (which includes the rent collector and pay master).

5.3 Lost and found objects

The charges list in the bank account contains all objects which are charged to an account, including those a user may have lost track of. In most cases the user will want to remove such objects. This makes it important to be able to distinguish between "lost" and "known" objects.

User-level naming in Mungi is based on the Plan 9 naming service [Presotto et al., 1991], which allows users to tailor their own name spaces. There is no system-wide human-readable name for objects, lost objects in this context are those not appearing in a user's name space. In order to identify them the name service supports not only a mapping from human-readable names to object addresses, but also the inverse mapping. A user can therefore run a library function converting the object references in the charges list back into textual names *bound to that user's name space*. Objects for which this inverse mapping fails are lost and can be removed.

5.4 User operations

Typical user operations on bank accounts are modification (or cancellation) of an account's income stream, and one-off transfers between accounts. The former are performed by the bank manager on behalf of a user who presents a valid capability to the source account (as in the case when the income stream was initially set up). In the case of a one-off transfer, a capability for the source account is required. Should the transfer of money without agreement by the recipient be considered unsafe, the bank manager could instead issue the sender of the money with a signed certificate which the recipient can present to have the money credited to his account.

Another important operation is the deletion of (lost) objects drawing rent on an account. While the protection system will not let a process perform any operations on an object to which that process has no capability, it is important that the owner of a bank account can delete objects for which she is paying. Therefore the bank manager system will, on request from a process presenting a read capability to an account on which a particular object's rent is drawn, delete that object. The bank manager can do this, since it holds a read capability to the OT and therefore has full access to all objects. Note that this does not in any way bypass Mungi's protection system, or users' views thereof. Logically, when creating a new object (and presenting a bank account capability to which rent will be charged), the system records a delete capability to the new object in the bank account. The same effect can be achieved more efficiently as outlined above, so that such recording of object capabilities in a bank account is not necessary, but users are still aware that a read capability to a bank account implies delete capabilities to all objects whose rent is charged to that account.

6 Conclusions

We have presented Mungi's bank-account-based resource accounting system. Its main advantage is that it operates off-line, so that operations on objects (such as creation, deletion, and initialisation) are not slowed down by accounting.

An accounting system must be trusted as some of its operations are inherently privileged. Therefore, in a system where accounting is left to the providers of services, these servers must be trusted too (as in the case of Amoeba). We have gone the other way by providing accounting for storage, one of the basic resources, as part of the system. This can then be used to implement services at user-level without any need for trust.

The accounting system presented in this paper provides a high degree of flexibility to the system administrator. While it automatically responds to resource shortages, the system administrator has significant control over how soon, and how strongly, the market forces, which keep the system operational, take effect.

As all policies are implemented at user level, they are easily extended or modified. It is, e.g., easy to deal with special cases, such as users enjoying particular privileges.

A prototype of the accounting system exists, however, its performance can only be assessed once a multi-user Mungi system is operational and supporting "real" users.

While particularly useful in Mungi, the model is not restricted to a SASOS and could, *mutatis mutandis,* be implemented in other systems.

References

Anderson, M, Pose, R, and Wallace, C. S (1986). A password-capability system. *Computer J.*, 29:1–8.

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 Tr. Comp. Syst.*, 12:271–307.

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

Heiser, G, Elphinstone, K, Russell, S, and Vochteloo, J (1994). Mungi: A distributed single address-space operating system. In *17th ACSC*, pages 271–80, Christchurch, New Zealand.

Heiser, G, Elphinstone, K, Vochteloo, J, Russell, S, and Liedtke, J (1998). The Mungi single-address-space operating system. *Software—Pract. & Exp.* To appear.

Lam, F. F (1995). Resource accounting in Mungi. BE thesis, Comp. Sci. & Engin., UNSW, Sydney 2052, Australia.

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

Presotto, D, Pike, R, Thompson, K, and Trickey, H (1991). Plan 9, a distributed system. In *Spring EurOpen Conference*, pages 43–50, Tromsø, Norway.

Vochteloo, J, Elphinstone, K, Russell, S, and Heiser, G (1996). Protection domain extensions in Mungi. In *5th IWOOOS.*, pages 161–165, Seattle, WA, USA. IEEE.

Waldspurger, C. A and Weihl, W. E (1996). An object-oriented framework for modular resource management. In *5th IWOOOS*, pages 138–143, Seattle, WA, USA. IEEE.