M. I. T. Laboratory for Computer Science
Computer Systems Research Division

15 March 1976
Request for Comments No. 109

THE QUOTA PROBLEM ON MULTICS

by Philippe Janson.

This short RFC attempts to describe the quota mechanism in NSS to define the quota problem and to propose solutions for it.

This is an informal working paper of the M. I. T. Laboratory for Computer Science, Computer Systems Research Division. It should not be reproduced without the author's permission, and it should not be referenced in other publications.

0. Definitions.

0.1. Quota cells.

A data structure called a quota account is associated with every directory of the file system. A quota account could be defined by the following declaration:

```
dcl 1 quota_account,
    2 master_dir_sw,
    2 quota_cell (0:1),
    3 received,            /* R */
    3 quota,               /* Q */
    3 used,                /* U */
    3 time_record_product,         /* TRP */
    3 time_product_updated;        /* TUP */
```

As can be seen, every quota account contains two quota cells (QC). QC(0) is used to account for directory pages. It is not maintained in the current implementation of NSS. I will further ignore it. The only QC I will be talking about is QC(1), which is used to account for non-directory pages.

0.2. Quota operations.

There exist four distinct quota operations:

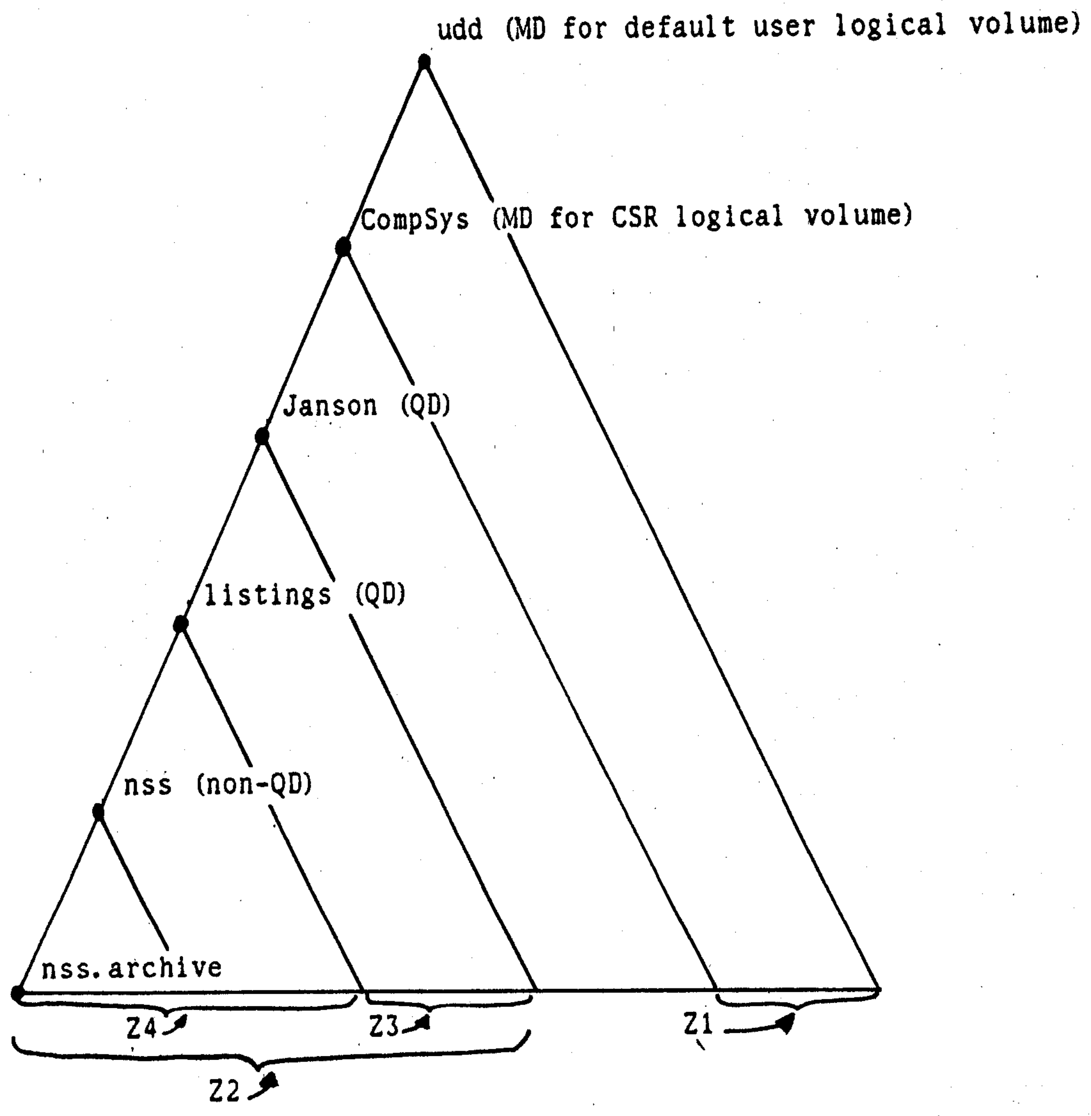
-- change_used (CU), which affects the U, TRP and TUP fields of a chain of hierarchically related QCs;

-- reset_usage (RU), which affects the TRP and TUP fields of one QC;

-- move_quota (MQ), which affects all fields (except the master_dir_sw) of two QCs that must be parent and son;

-- set_quota (SQ), which affects all the fields (except the master_dir_sw) of a chain of hierarchically related QCs.

figure 0: example of quota hierarchy to be used in the following sections


udd (MD for default user logical volume)
CompSys (MD for CSR logical volume)
Janson (QD)
listings (QD)
nss (non-QD)
nss.archive
Z4
Z3
Z1
Z2

0.3. Master directories.

Any directory for which the master_dir_sw is set is called a master directory (MD). MDs are designed to control the usage of resources on logical volumes. The registration data of every logical volume specifies what directories are MDs for this logical volume and how much space can be used by the segments below each MD. Only the MDs for a logical volume and non-MDs below these can store their inferior segments in the logical volume. The R field of the QC of a MD can never be subject to any operation. R is defined once and for all by information stored in the registration data of the logical volume for which the directory is a MD. For a MD, R indicates the (fixed) portion of the physical resources represented by the logical volume that can be used by segments in the subtree rooted at the MD less all subtrees rooted at MDs -if any- below the MD under consideration. For instance, R(udd) is the amount of physical resources that can be used by the segments in zone Z1. Notice that every MD is a quota directory (and is treated as such) because...

0.4. Quota directories.

...quota directories (QD) are recognized by their R field being non-null. For a QD, the R field indicates very naturally the amount of storage that can be used by segments in the subtree rooted at that QD less the subtrees rooted at MDs -if any- below the QD. For instance, R(Janson) indicates the amount of storage available for segments in zone Z2, since there is no MD below Janson. The Q field of a QD indicates the amount of resources that can be used for segments in the subtree rooted at that QD less the subtrees rooted at QDs -if any- below the QD under consideration. For instance, Q(Janson) is the amount of storage available for segments in zone Z3.

0.5. The CU operation.

The CU operation cannot be invoked directly by users. It is invoked as a result of touching a null (empty) page or truncating a segment or deactivating a segment containing allocated null pages (zero pages). In other words, it is invoked when a user causes a segment to grow or to shrink. Let us examine the case of growing a segment. The shrinking case is symmetric. Assume segment nss.archive is to be grown in zone Z4 (figure 0). The CU operation performs as follows:

(1) walks up the hierarchy until finds a QD (listings);

(2) authorizes growing only if U(listings) < Q(listings);

(3) updates the QC of listings: TRP = TRP + (time - TUP) * U (integration)
TUP = time;

(4) adds one to the U field of every directory from the QD down to the parent directory of the segment to be grown (listings and nss).

Thus, at any time and even for non-QDs, the U field represents a running account of the storage used in the subtree rooted at that directory less all subtrees rooted at inferior QDs -if any. For a QD, the TRP is the current value of the integral of the discrete plot of U versus time. Notice that the U field of a non-QD is useless as far as accounting is concerned since the associated TRP integral is not maintained but it is used by the MQ and SQ operation for efficiency. Keep this fact in mind for later use.

0.6. The RU operation.

The RU operation is privileged and can be invoked only by system administrators to collect accounting information and bill users periodically. For every QD, RU returns resources_used = TRP + (time - TUP) * U and then sets TRP = 0 and TUP = time.

0.7. The MQ operation.

The MQ operation can be invoked by users to move a (positive or negative) amount of Q between a QD P and one of its sons T, which is regarded as the target of the operation. The user requesting the operation must have modify access to both directories to perform the operation. By moving Q from Janson to listings, one does not change the amount of resources that can be used by segments in zone Z2. One only indicates the desire to partition the amount R received by Janson into a portion Q(Janson) to be used in zone Z3 and a portion Q(listings) to be used in zone Z4. Assuming the user has modify permission to directories P and T, and P is a QD, the MQ function moves a quota Qo as follows (T need not be a QD):

```
if Qo > 0
then do;                                                /* downward move */
        if (Q(T) = 0) & (U(T) < Qo) & (U(P) - U(T) < Q(P) - Qo)
        then do;                                        /* make T a QD (*) */
                TRP(P) = integral;
                TUP(P) = time;
                U(P) = U(P) - U(T);
                TUP(T) = time;
                go to move;
            end;
        if (Q(T) > 0) & ( U(T) < Qo + Q(T)) & (U(P) < Q(P) - Qo)
        then go to move;
        return error;
    end;
```

```
if Qo < 0
then do;                                            /* upward move */
        if (Q(T) + Qo = 0) & (Q(T) = R(T))
        then do;                                    /* make T a non-QD (*) */
                TRP(T) = integral;                  /* charges to be moved */
                TRP(P) = integral + TRP(T);
                TUP(P) = time;
                U(P) = U(P) + U(T);
                TUP(T), TRP(T) = 0;
                go to move;
            end;
        if (Q(T) + Qo > U(T))
        then go to move;
        return error;
    end;

move:   Q(P) = Q(P) - Qo;
        Q(T) = Q(T) + Qo;
        R(T) = R(T) + Qo;  return;
```

0.8. The SQ operation.

The SQ operation is a privileged operation that is restricted to system administrators' use. It forces the Q of an arbitrary directory T to become a given value Qo. The SQ operation is always authorized and operates as follows:

(1) if R(T) = 0 then walks up the hierarchy until finds a QD, subtract U(T) from the U field and update the TRP and TUP fields of every directory encountered up to and including the QD. (This is necessary to be consistent with the definition of a QD.)

(2) R(T) = R(T) - Q(T) + Qo. (This is necessary for subsequent MQs on T to be consistent.)

(3) Q(T) = Qo.

Notice that, unlike the MQ operation, the SQ operation exhibits two peculiar features (bugs?). First, it may force the traget QC in a state where Q(T) < U(T). Second, it may cause a directory T to become a QD while its parent is not a QD, which is impossible with the MQ operation.

1. Problems.

1.0. Implementation.

To understand what the quota problem is, it is necessary to understand how the mechanism is implemented in NSS and why it is that way.

The hierarchy of QCs is the same as the directory hierarchy and is therefore managed by directory control (DC). This is because the functionality of the Multics quota mechanism was defined as such. (Otherwise a separate hierarchy could have been implemented.) Quota operations involving hierarchy considerations require knowledge that is embedded at DC level.

QCs are stored in directory vtoces, i.e. in segment control (SC) level data bases. This is for efficiency reasons. A QC requires only a few (say 8) words of storage. Therefore, implementing it as a stand-alone segment would be unacceptably expensive. This is the typical small object problem. The usual solution to the problem is to store the small object together with other small objects in one segment. In the old Multics storage system, QCs were stored in directories. In NSS, a QC is stored in the corresponding directory vtoce. This is more efficient because accessing a QC only requires that the directory be active and not that any of its pages be in core. As a way to find the QC a segment must be charged to but mainly for efficiency again, any directory of which one or more sons are active is kept active so as to reduce the delay that would be incured by the CU operation if it had to address a QC by pathname and wait for it to be activated.

Finally, ignoring hierarchy considerations, individual QCs are managed by page control (PC). This is only because Multics does not distinguish ordinary page faults from page faults on null pages. All faults are handled by PC. The CU operation is entirely implemented at the PC level.

As a consequence of the above implementation, PC depends on SC because the data it manipulates to manage quota is stored at SC level, and it depends on DC because it uses astes that are threaded after DC information about the directory hierarchy to implement the CU operation. SC depends on DC because it must pay attention not to deactivate certain segments (namely directories) based on directory hierarchy considerations. Furthermore, it explicitly calls DC to propagate quota usage information up the hierarchy when truncating or deleting an inactive segment.

1.1. Cause 1: hardware misconception.

The first cause of the upward dependencies is the result of a misconception in the basic hardware and the supporting software for handling the CU operation. PC should never be involved in quota management. QCs apply to groups of pages and PC should not be aware of the grouping.

It would be easy to patch the hardware so that null pages would be distinguished from other pages (different directed faults). Reading a null page should simply return a word of zeroes. This would save resources, save work and avoid a read on a null page to cause a write on a QC, which causes trouble with respect to the AIM *-property. Writing a null page would cause a quota fault. Control could then be transfered at an appropriate level but certainly above PC. This eliminates the upward dependencies originating from PC and due to quota management.

1.2. Cause 2: QC storage and functionality.

The SC-DC dependency must still be eliminated. A detailed analysis of this dependency reveals that it occurs only because

(1) the quota hierarchy is coupled to the directory hierarchy

and

(2) a QC is stored in the SC level vtoce of the corresponding directory.

2. Solutions.

All following solutions to the SC-DC dependency are based on the fact that they eliminate one or both of the conditions that cause the dependency.

2.0. Trivial but inefficient.

A first solution to the problem consists of removing QCs from the SC level by storing them as they were originally, i.e. in directories, and to reflect quota faults into DC. DC can store an uninterpreted entry pointer in the kste of every initiated segment. Thus, on a quota fault, it can invoke the address space manager to extract the entry pointer from the kste corresponding to the faulting segment number. This first solution would unfortunately be inefficient because a QC would in general be out of core and inactive when it is needed to perform a CU operation, which is precisely what the current NSS design avoids so nicely. In addition, this solution adds complexity to the quota mechanism because the CU operation does not happen as an atomic operation under the protection of the page tables lock and must expect cases in which the segment to be grown has already been grown or is currently deactivated.

2.1. Individual quota management.

This solution, which is due to Mike Schroeder, is based on avoiding causes (1) and (2) of the upward dependency. It consists of designing a dedicated quota manager (QM). QCs are not stored in directory vtoces or in directories themselves. They are collected in the QM's internal data bases that could be permanently active segments, for instance. (According to data collected on Multics, this appears to be feasible.) They are not related by the directory hierarchy. They are threaded together into their own hierarchy (which may be parallel to the directory hierarchy and contain "indirect" QCs to correspond to non-QDs). Even though the quota hierarchy may be the same as the directory hierarchy, it must be maintained separately so that quota faults do not require extracting hierarchy information form the DC level and the QM can operate independantly from DC. Every time a directory is created, DC asks QM to create a corresponding QC and stores the name of that QC in the directory. Each time a segment is appended to a directory, DC calls SC to store the name of the appropriate QC in the vtoce of the new segment, as uninterpreted information. Quota faults are reflected into the QM. The QM can thus extract the identity of the QC to be charged from the vtoce of the faulting segment by invoking SC. It can then operate on that and related QCs as dictated by the threads.

This solution may be somewhat less efficient than the NSS implementation because of the posibility to take page faults on the QM data bases. This minor problem could be avoided by using a cache (wired segment) to contain the most recently used or the currently "active" QCs. Such a design may save some space in the AST since QCs are not there any more and parent directories need not be kept active.

2.2. Constrained hierarchy changes.

This final solution, which I have designed for my thesis, is based on the elimination of cause (1) of the upward dependency. It may sound unattractive because it implies a modification of the functionality of quota. In fact, this modification has very minor effects because only a few (a dozen) users are in a position to notice it, and these users might never notice it anyway.

The reason for this transparency is that the modification affects only owners of QDs. And aside from project and user directories for which the modification would never be felt anyway, only a dozen users (according to data collected on Multics) have subdirectories that are QDs.

The modification stems from an observation that the CU operation would never require any hierarchy consideration if it were not for the parallel existence of MQ and SQ operations. If every segment vtoce contained a name directly identifying the QC of the QD the segment is to be charged to, the CU and RU operations could still work and would not use any hierarchy. Only SQ and MQ require knowledge about the quota hierarchy. Thus, CU could be supported entirely within SC and cause no upward dependency. However, two problems would have to be solved. First, when a MQ or a SQ operation would cause a QD to become a non-QD or vice-versa, the QC identifiers stored in the vtoce of every segment in the subtree rooted at the directory affected by the change would have to be updated. Second, since the U fields of non-QDs are no more maintained, it would be necessary to explore the whole the subtree to compute its U field in the case where a non-QD becomes a QD. Not only would these two procedures be unacceptably expensive but in order to preserve the proper synchronization between SQ/MQ operations and CU operation, it would be neceary to perform these operation under the protection of a lock that would prevent anybody from growing or shrinking a segment while a SQ/MQ operation is in progress, and this may take a while. The lock is needed to guarantee that a segment is not charged to the wrong QC or that that cell is not charged to the right QC before that cell is properly initialized. (Think about this or see me.)

To avoid these problems and yet be able to store into each segment vtoce the identifier of the QC of the appropriate QD, I constrain the variations of the quota hierarchy so that a directory can change from Q to non-Q status or vice-versa only if it is empty (no branches), i.e. if there is no need to update any QC pointer. As mentioned earlier, this change does not affect most QDs, in particular project and user directories, because these are made QDs while they are still empty and loose their status only when they are deleted.

With this design, SC can be made independant of DC. Since the QC to which a segment must be charged cannot change, SC can keep it active on the basis of <u>segment</u> usage information and <u>not</u> on the basis of any <u>directory</u> hierarchy considerations. QCs may still be stored in directory vtoces. However, their active version does not have to be in the AST. It may be in an AQT cache. In any case, this solution eliminates the upward dependency, saves as much space in the AST as Mike Schroeder's solution and simplifies quite a bit the quota mechanism.