Contents

Introduction ........................................................................................................................................ 3
Disclaimer ........................................................................................................................................... 3

Series Content ............................................................................................................................... 4
Intended Audience .......................................................................................................................... 4

UniVerse Configuration .................................................................................................................. 5

Changing Parameters .................................................................................................................... 6
IMPORTANT - Back up your Configuration! ................................................................................. 7
Restoring the Configuration ......................................................................................................... 8

MFILES and the Rotating File Pool ............................................................................................ 9
T30FILE and Dynamic Files ........................................................................................................ 11
UVTEMP - Temporary Space ......................................................................................................... 13
SELEBUF - Select Buffer ................................................................................................................ 14
TXMEM - Transaction Cache .......................................................................................................... 15
GSEMNUM, GITABSZ, RLTABSZ - Lock Table ......................................................................... 16
FSEMNUM, FLTABSZ, RLTABSZ, MAXRLOCK - File Lock Tables ............................................. 20
Finishing Off ................................................................................................................................. 21
Introduction

As expected for an application platform called upon to provide services across a wide range of scenarios, from small independent end user sites through to large scale solutions, UniVerse requires a certain degree of configuration management.

The UniVerse configurable parameters, visible through the CONFIG ALL command, can be grouped into four categories:

- Those that enable or disable specific features.
- Those that modify behaviour to emulate other platforms.
- Those that affect run-time performance.
- Those that determine hard limits for the system.

With the assimilation of more features from UniData in version 11.x, the number of parameters that fall into the last category has increased, and there is some confusion over the role of these parameters in UniVerse.

This two-part blog will concentrate on the latter two categories, explaining some of these parameters and when and where you might choose to adjust them.

Disclaimer

Please note that these parameters can fundamentally affect the way in which UniVerse and your applications will run, and should only be adjusted with caution. Configuration changes should be tested and Brian Leach Consulting Ltd. and Rocket Software Inc. will accept no responsibility for any damage or performance loss arising from your choice of parameters. If you are unsure of the changes you intend to make, seek professional advice.
Series Content

This series will highlight and attempt to explain those UniVerse configuration parameters that directly affect performance and that create hard limits on your systems.

The first post will concentrate on those parameters that have formed the base line for UniVerse prior to version 11. These are more generally understood, and so this acts to consolidate the information that is generally available but scattered.

The second post will look specifically at the new parameters added at version 11, predominantly those that have been imported from UniData. We will look at the junction between these parameters and their new role: how they are historically used in UniData, and how they are now applied to UniVerse.

Intended Audience

This series is pitched at experienced systems administrators and DBAs, and a working knowledge of UniVerse is assumed throughout.
UniVerse Configuration

Not all UniVerse sites are equal. If you consider the broad range of UniVerse applications, from small in-house systems to large cloud-based solutions; and the huge disparity in what those systems are asked to perform, from complex financial and engineering calculations through to simple (if high volume) retail transactions, it is surprising just how well UniVerse can adapt with minimal tuning.

Aside from a couple of (hopefully) well-known hard limits, it has been historically possible to consider UniVerse as practically ‘load-and-go’ software: although performance may be compromised, without tuning it has still been possible to have a running application. The growth in hardware performance has had its part to play: a modern system using SSDs for storage will massively outperform servers that are only a few years based on older, HDD technologies; and operating systems and virtualization options continue to develop in reach and performance.

However, to gain the most benefit from your UniVerse installation, an understanding of the role played by the configuration parameters is still important. For larger sites running high number of concurrent processes, an understanding of how the new version 11 parameters apply is essential. UniData DBAs have long been accustomed to the need to tune their systems to have them work reliably: whilst externally similar the UniData architecture is very different than UniVerse and for all but the most trivial applications UniData is not load-and-go.

Many performance problems are still, of course, down to application design. If your application is poorly optimized, platform performance is only ever going to be a sticking plaster, and the application level is the still the first port of call.
Changing Parameters

The UniVerse parameters are surfaced through the `uvconfig` file, resident in the UniVerse installation directory. This file is in text format, with each parameter prefixed by a short description of varying levels of helpfulness.

If you open the `uvconfig` file in your favourite editor (or failing that, log to UV and type ED &UFD& uvconfig), you will see a layout similar to the one below:

```plaintext
# MFILES - specifies the size of the UniVerse rotating file pool. The value of MFILES should be set to a value no greater than the kernels per process open file limit less the sum of the maximum number of named pipes opened by a user application and the 8 files reserved for internal UniVerse use.
MFILES 200
```

Whilst the `uvconfig` file is used to maintain these parameters, it is not where UniVerse reads them from. The runtime parameters, along with additional details related to authorization, live in the hidden `.uvconfig` file. This is a binary file, non-editable and interrogated as part of the UniVerse start-up procedure.
IMPORTANT - Back up your Configuration!

Before making any configuration changes, you should back up both the uvconfig and the .uvconfig files so you can quickly restore them should something go awry.

Under Windows, you cannot use the regular Windows explorer to do this - it will complain about the file name. And, of course, don’t use the UniVerse COPY command to back up the binary .uvconfig file.

The old-style Windows COPY command is similarly unimpressed:

```
C:\U2\UV>attrib .uvconfig
A   H       C:\U2\UV\uvconfig
C:\U2\UV>copy .uvconfig .uvconfig_20171002
.uvconfig
The system cannot find the file specified.
0 file(s) copied.
```

To make the copy you need to remove the hidden attribute.

```
C:\U2\UV>attrib -h .uvconfig
C:\U2\UV>copy .uvconfig .uvconfig_20171002
1 file(s) copied.
```

Once you have your backup of the two files, and have modified the uvconfig file for your purposes, to apply this you can use the uvregen command:

```
C:\U2\UV>bin\uvregen
uvregen: reconfiguration complete, disk segment size is 5329600
```

This will not take effect until UniVerse is restarted.

```
C:\U2\UV>bin\uv -admin -stop
uvtelnet service stopped successfully
C:\U2\UV>bin\uv -admin -start
universe service start pending...
universe service started successfully.
uvtelnet service start pending...
uvtelnet service started successfully.
```

It goes without saying that you should not restart UniVerse unless all processes - user and background - have been logged out.
Restoring the Configuration

If you need to back out the configuration following a change, you can copy back your saved .uvconfig file (again, after stopping UniVerse). On Windows, the regeneration will have marked this as a hidden file once again, giving a misleading Access Denied message:

```
C:\U2\UV>copy .uvconfig_20171002 .uvconfig
Overwrite .uvconfig? (Yes/No/All): Y
Access is denied.
0 file(s) copied.
```

As before, you just need to remove the hidden attribute before copying over:

```
C:\U2\UV>attrib .uvconfig
A   H        C:\U2\UV\uvconfig
C:\U2\UV>attrib -H .uvconfig
C:\U2\UV>copy .uvconfig_20171002 .uvconfig
Overwrite .uvconfig? (Yes/No/All): Y
1 file(s) copied.
C:\U2\UV>bin\uv -admin -start
universe service start pending...
universe service started successfully.
uvtelnet service start pending...
uvtelnet service started successfully.
```
MFILES and the Rotating File Pool

UniVerse applications are highly file-based. Unlike most SQL databases, UniVerse does not explicitly cache its changes in memory, leaving file caching up to the operating system. To see the effect of file caching, run a select on a file and time the result. Then run the same select a second time: you may see a very different performance.

Opening files is expensive. With today’s SANs and SSD storage, this is less so than in the past, but even so it represents an overhead on the system and historically UNIX systems placed hard limits on the number of files a single process could open (man NOFILES). UniVerse, coming from a UNIX background, therefore also limits the number of files that can be opened concurrently, even though these limits are of little concern to most modern UNIXen.

The rotating file pool represents a transparent layer between the application and the underlying OS. An application need not concern itself with any limitations on the number of open files, and many MultiValue applications simply open all major files at initialization, holding file variables in common or named common. The rotating file pool was created to manage this discrepancy.

You can see the effect of the rotating file pool by choosing a UniVerse process and running the PORT.STATUS PORT n FILEMAP command (in the uv account).

The rotating file pool is set up inside a uv process. It holds an array of operating system file pointers that are swapped in and out on a last-used basis if the number of files opened by the application exceeds MFILES.
Over time the default size of the rotating file pool has changed on new UniVerse installations. Back in the 1990s it was as low as 8 or 12 (!) but now stands at a more acceptable 200. However, whilst that may seem a reasonable per-process limit, as new features have also been added to UniVerse and others become more solid and widely used that is not quite the whole story.

It is important to realize that these are not UniVerse file pointers, they are operating system files, and there is not a one-to-one relationship between these. Consider the following scenario: a user runs a SELECT command against a dynamic file containing several secondary indices: that SELECT in turn uses dictionaries that perform XLATES or T-Correlative lookups to another dynamic file. From the operating system perspective, this means:

- Two file pointers for each dynamic file data portion (DATA.30 and OVER.30).
- One for each dictionary.
- One for every one of the secondary index files, along with the index map - whether used or not they get opened automatically with the file.
- One for the select list.

So a simple SELECT command might easily consume at least a dozen file pointers. For a distributed file the numbers multiply, especially where the part files also contain secondary indices. Suddenly that default limit of 200 doesn’t look so large after all.
T30FILE and Dynamic Files

The MFILES parameter is a performance setting. T30FILE is the first parameter that introduces a hard limit - exceed this and your application will probably fail.

You should already be familiar with dynamic files, their advantages and their pitfalls. Dynamic files constantly readjust their modulus and redistribute their content to optimize storage and retrieval: a dynamic file will never perform as well as a perfectly sized static hashed file, but should never perform as poorly as a badly sized static hashed file. Dynamic files spread the pain of administration over the lifetime of the file.

This ease of administration comes at a cost. Dynamic files are more volatile and more limited in what you can do. They developed from theoretical models on linear hashing that have a background in compiler generation - but those models were never intended for multi-process use and so a great deal of work has been undertaken over the years to make these stable and efficient. UniData also has dynamic files, but the arrangement and management of those is internally wholly different.

In order to achieve this whilst retaining some measure of performance, UniVerse tracks the current file statistics for as long as a dynamic file is open. Unlike other file types that can simply be opened by individual processes, this must be coordinated through the global shared memory segment. Specifically, UniVerse tracks dynamic control information including the current load factors and statistics, the current modulus and the split pointer, hashing algorithm and large record size: details that are eventually stored in the file header.

Whilst the file is in use, to prevent excessive write-backs and to equally prevent other processes from re-reading the file header before every access, UniVerse updates these within the shared memory segment: the values are written back to the file header when the file is closed or when certain values held in the T30FILE table are updated (for example, the current modulus or the next split pointer).

These statistics are held in a fixed size array determined by the T30FILE parameter, which therefore sets an absolute limit to the number of dynamic files that you can have open on your system at any one time. If there is no available slot for a type 30 file, the open will fail. Once all processes have closed a file it will be removed from the array but if your application holds files open in named common, for example, they will persist until no longer referenced.
The T30 file list can be viewed using the ANALYZE.SHM -d command: this can give an indication of how much head room is available in this array, but only as a snapshot. In my experience sites usually only look at T30FILE once they have hit a stop situation. Another limitation is that the T30FILE structure is protected by a single semaphore and that can also act as a bottleneck if very large numbers of dynamic files are used.

So dynamic files are useful but need to be managed carefully and within the confines of the T30FILE array. Reducing the number of concurrently accessed dynamic files may often be a better strategy than dramatically increasing T30FILE.

Consider, for example, the case of a distributed file holding transaction data separated by periods - a not uncommon use for distributed files and one that makes archival easy (after a regulatory or similar period the older part files can be archived and removed). Whilst this is being built and modified, the part file representing the current period might best be served as dynamic, but once the data is settled and historic periods are retained primarily for archival purposes, or are only lightly modified and can be sized based on their content, now perhaps they should be resized to static files. Not all part file need to share the same file type.

In other cases, this particularly affects sites that create dynamic files as temporary work files because they cannot predict in advance how much data a given job will generate in-flow and thus do not want to use static files that would either waste lots of space through being oversized or end up massively undersized.

Often a better choice is to use the underrated type 25 files for this in situations where multi-user contention - locking issues are the downside of B+trees - is less of an issue. Where a system is under pressure and is using work files for heritage processes that require temporary storage of records, it is also worth considering the new data structures available that transcend the historic limitations of dimensioned and dynamic arrays, especially through Universe Dynamic Objects (UDOs) or python, and whether these allow the need for such work files can be engineered out of a solution.
UVTEMP - Temporary Space

Large select lists, out-of-cache transactions, captured output, compiles - the UniVerse temporary space is constantly being hammered, although some of these can be reduced by attending to configurations we will meet later in this post.

By default, the temporary space is held within the uv account in a directory called, suitably enough, uvtemp. Granted that you have probably placed your UniVerse installation on a nice, fast local SSD that may be good enough, but there may be optimisations from moving this to other storage. As the items stored in here are transient, there is no reason for them to ever hit disk so long as there is capacity for them to be held in memory.

Some modern UNIX platforms automatically map their temporary space (/tmp) into memory, usually with a fall-back should some threshold - either fixed or dynamic based on availability - be exceeded. Similarly, on Windows you can create a RAM drive and associate that with a temporary folder, though most RAM drive software places a hard limit on this.

In any case, if you find your application is generating large numbers of selects, for example, there may be mileage in ensuring that your UVTEMP is held in the fastest location possible, particularly on Windows where file cache management is more aggressive at releasing pages than on UNIX.

Better still is to reduce the number of select lists that need to be written to temp space in the first place, and that we will consider later in this post.
SELBUF - Select Buffer

The UVTEMP parameter determines where UniVerse will store temporary select lists that exceed a threshold, but writing these out is obviously a performance hit best avoided. For this you can tune the SELBUF parameter.

SELBUF sets the largest size of a select list that will be stored in memory before UniVerse elects to write it out to disk. Unlike many of the other parameters for which the defaults have been increased over time, this has not really kept pace with increasing hardware capacity and remains at a very lowly 4K. That’s not a lot of record keys!

Increasing SELBUF can be a relatively low impact way to improve performance, particularly for applications that make heavy use of selections. The only easy way to track the likely sizes is to watch your temporary space and make an estimate based on the median size of the temporary select lists that get written away, but if you have the capacity you can bump this up by many orders of magnitude.

SELBUF is a threshold, but for creating temporary select lists UniVerse still appears to assign memory dynamically, so in most cases you can consider setting this to a large value without running out of resources.
**TXMEM - Transaction Cache**

A similar story is attached to TXMEM. This assigns a local buffer for transactions. To clarify what has been a source of confusion through a somewhat tortuous history - you do **not** need to be running transaction logging to safely use transactions.

UniVerse transactions, whether explicitly created through the business language or automatically through SQL statements and the use of triggers, will use this as a scratch buffer for holding transaction level changes up until the point they are committed or rolled back.

If your application typically makes use of large transactions, the default of 32K may again be somewhat impoverished. One thing that that the uvconfig description does not make clear, and that may cause some sysadmins sleepless nights, is that the TXMEM is not an absolute limit. Exceeding TXMEM for a transaction will not cause that transaction to fail.

In situations where TXMEM is exceeded, as with temporary select lists the excess is written to the temporary directory. Of course, this again adds an overhead so tuning TXMEM is recommended on transactional systems. This is highly application specific so will really depend on other, business level metrics to determine an appropriate size.
GSEMNUM, GLTABSZ, RLTABSZ - Lock Table

Where UniVerse sites typically do look at the configuration parameters, it is generally around the lock table. This is another hard limit on the system: every novice developer will probably experience the pain of accidentally filling the lock table in a ReadU loop - just once. Every system has it’s Achilles heel and the lock table is still the best way to stop a UniVerse system in its tracks.

In its defence, the lock table was devised in the days of smaller user numbers and before the advent of SQL semantics. The isolation level requires for SQL statements and for actions that take place under those same semantics - transactions and triggers - force the creation of locks to protect all updates. This includes setting record locks for new record creation, something heritage applications would often forgo. The isolation levels have placed an additional burden on the lock table that could not have been foreseen when it was originally created. If you are using transaction semantics, all record locks remain active until the COMMIT: so in a loop doing a regular READU and WRITE, the WRITE will not release the lock as it would without transaction semantics.

Additionally, the lock table was designed at a time when semaphores - which provide the key role of protecting access to the lock table - were also a very limited resource on UNIX. As with most of the other UNIX kernel parameters, such a limitation is rarely the case today.

The configuration parameters GSEMNUM, GLTABSZ and RLTABSZ work together to define the shape and size of the lock table, which is another central resource held in the global shared memory segment. To understand how these cooperate requires a little more background than the other parameters we have covered to date.

You should already have some understanding of the two aspects of the lock table: transient group level locks used to coordinate access to pages in hashed files, and business level locks to restrict update access to records and files.

\[1\] Windows does not have the same concept of shared memory as UNIX, but this is emulated through a block owned by the UniVerse Resource service.
These are both rather neatly expressed in the output from the familiar LIST.READU command:

**LIST.READU EVERY**

<table>
<thead>
<tr>
<th>Active Group Locks:</th>
<th>Record Group Group Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device.... Inode..... Netnode Userno Lmode G-Address. Locks ...RD ...SH ...EX</td>
<td></td>
</tr>
<tr>
<td>1974183665 403854372 0 2860 28 RD 7400 0 1 0 0</td>
<td></td>
</tr>
<tr>
<td>1974183665 403854372 0 2580 75 IN 3E00 1 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Active Record Locks:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device.... Inode..... Netnode Userno Lmode Pid Item-ID.......................</td>
</tr>
<tr>
<td>1974183665 403854372 0 2580 75 RU 2580 10</td>
</tr>
</tbody>
</table>

This listing largely parallels the organization of the lock table itself.

The lock table itself is composed of two hashed structures: the group lock table and the record (update) lock table. Both are protected using the same set of semaphores.

When a process requires transient access to a file group, either to read or write the content, it requests a read or write lock at the group level based on the primary group to which the record hashes: not any overflow groups, that are considered a logical extension - the group locking needs to ensure that all of these remain logically intact as a unit. Badly sized files are therefore one of the main culprits to poor locking, since the process must retain the lock for longer whilst managing the overflow chains.
The group locking ensures that there is segregation between processes accessing hashed files, whilst still allowing concurrency of access for processes targeting different primary groups. There is no need for UniVerse to lock the whole file during an update. Non-hashed files are managed by the lock table as though they had a single, zero group, which again creates a potential bottleneck and is one reason why holding data in directory files - unless explicitly required by the application structure - is generally a bad idea as it can lead to excessive blocking should large numbers of users need to access the data concurrently.

As with UniVerse file storage itself, the lock management first hashes the request - including the inode that uniquely identifies the file and the target group - against the number of semaphores that determine and protect the rows of the lock tables (GSEMNUM). This hash result identifies the specific row number in which UniVerse will attempt to locate a free slot for the group lock entry. This row number is shown in the Lmode column of the LIST.READU group lock table.

The process will attempt to gain access to that row, but if another process owns the semaphore it must sit and wait, retrying after a nominated period, until the semaphore is released. You have probably experienced the delay when waiting for a record lock that you know has just been released.

The number of such collisions and the number of retries before the process can get hold of the semaphore are both recorded in the ANALYZE.SHM -s or SEMAPHORE.STATUS command: if the number of collisions is high and especially if there is also a high disparity between the collisions and the number of retries required to resolve the collision this suggests that the semaphores are becoming a major bottleneck for the application.

When a process requests a business level update or shared read lock (READU or READL) the lock management again first uses the group lock table and places an IN (INformation) lock in the requested row. This acts as a first stage check: any process looking to see whether a record is locked can first examine the group lock table to see whether the information lock exists for that group key. The group lock table therefore has two distinct uses.

Once through the group lock table, a business level lock set or test will use the same row in the ReadU Lock Table to hold the item level lock so that it can be handled in the same semaphore action. This records information about the network id, device id, inode, group and item id of the record in question, and includes the pid holding the lock. It is possible though extremely rare to get ‘phantom’ locks since not all of the key is stored.
All this means that the whilst there is a theoretical maximum number of records lock slots available as GSEMNUM * RLTABSZ (number of rows and number of columns in the readu lock table) that assumes all records hash evenly into the group lock table. If you examine a LIST.READU listing on your system you may well discover that this distribution is uneven, just as with a badly distributed hashed file.

In practical terms, if you are bouncing up against the lock table limits, you have two choices: to increase the number of rows or the number of slots for each row.

In terms of performance, the likelihood of collisions occurring when trying to access a row of the lock table is dependent on the number of rows (GSEMNUM) and on the size of each row - which in turn determines the amount of time a semaphore is held whilst UniVerse scans through a row. This should optimally be within one time slice whilst the semaphore is held, or there can be a significant effect on the CPU. In most cases, therefore, it is probably better when increasing the lock table to give preference to extend the number of rows rather than the number of slots in each row.
FSEMNUM, FLTABLES, RLTABLES, MAXRLOCK - File Lock Tables

Working in conjunction with the group lock tables are the file lock tables. These use a separate series of semaphores and sizing based on the FSEMNUM parameter.

File locks are used by processes that need to control the state of a file as a whole. At the business level, the FileLock and FileUnlock commands can be used to explicitly set and release file level locks, and may be preferable in some cases to holding large numbers of individual record locks.

File locks are also set by various database commands, including the CLEAR.FILE command and the associated ClearFile statement in the business language, resizing and other restructuring operations. The MAXRLOCK parameter can also come into play here: if a SQL statement requires a large number of locks this sets the threshold at which it will switch to using a file lock to prevent overwhelming the lock table.

You can see the file locks as an additional section in the LIST.READU EVERY command, although it is rare for it to show.

Active File Locks:
Device.... Inode..... Netnode Userno   Lmode         Pid
1880103218 9254996977       0      2    2 FX        6240

The file lock table runs along similar lines to the group lock table: the file identifier is hashed into the FSEMNUM array to determine which row will hold the details, and the process will obtain the equivalent semaphore.
Finishing Off

This concludes the first part of this series into UniVerse Configuration.

This post has walked you through some of the configuration parameters that can affect performance or capacity of your UniVerse application, focusing on those that have formed part of the UniVerse environment before release 11.x.

In the next post, we will be looking at the new configuration parameters, what they mean and why you may need to adjust them.