Advancing with UNICOS Toward IRIX

Barry Sharp
Boeing Shared Services Group - Technical Services
High Performance Computing - Engineering Operating Systems
The Boeing Company, P.O. Box 3707 MC 7J-04, Seattle, WA 98124-2207, USA
barry.sharp@boeing.com

Abstract

During the past eight years of running UNICOS on X-MP, Y-MP, and T90 systems numerous enhancements to the kernel, System Administration and User-Level facilities have been made. This paper outlines the rationale for these and provides descriptions on their implementation. The intent is to share this inventory with other UNICOS member sites and to suggest those that might be applicable to the Origin2000 IRIX system.

1.0 Introduction

Life started in 1979, which now appears to be in the murky distant past, with a Cray-1S having just 1/4 Mwords of memory and COS 1.08. The good old days some would say. Much was needed at this time in terms of adding value to the operating system and supporting infrastructure software. By 1989 we had reached COS 1.14 on an X-MP and were aspiring to a Y-MP8E running UNICOS 5.0. Our COS 1.14 system contained around 500,000 lines of local modifications by this time, mostly for handling connectivity issues to other non-Cray computing platforms such as CDC and IBM machines. The Y-MP/UNICOS system negated the need for such local modifications and most of it was therefore destined for the bit-bucket. However, UNICOS was not utopia by any means. The remainder of this paper describes how we have advanced with UNICOS toward IRIX in terms of what we have added to support our systems since the UNICOS 5.0 days to the present day with a T916/16512 under UNICOS 10 and an Origin2000 under IRIX 6.4.

Several of our UNICOS enhancements are, in our minds, applicable to the Origin2000 IRIX system. If your site concurs I strongly encourage you make your voice heard at CUG and in your future discussions with SGI/Cray.

2.0 Our History

To establish our credentials a brief history of our major software projects over the span of some 18 years is given below.

- In 1979 we installed our first Cray machine; a CRAY-1S running under COS 1.08. The majority of local modifications were related to connectivity to non-CRAY computers

for job submission via CDC Station software and data transfers, permanent dataset enhancements (including security such as ‘permits’), data backups, and resource accounting.

- Early ‘80s we upgraded to an X-MP running under COS 1.12. Major tasks while we evolved with COS were related to porting all local modifications, collaborative effort with Cray to extend the Permanent Dataset Management Task (DSC Extensions - DXT) and for dealing with the first Cray delivery of a128 Megaword SSD (at the time COS 1.12 only supported a max SSD size of 64 Megawords). We developed a system-wide Freeze/Thaw facility allowing QA testing which protected production work.
- 1986-87 Ported COS 1.13 to an SCS-40
- 1988-89 Ported UNIX V5 to an SCS-40XM; named SCENIX - a real fun project
- 1990-92 Contracted with NEC to port COSMIC NQS and UniTree to an SX-3
- 1989-1998 - Advanced with UNICOS on X-MP, Y-MP, T94 and T916

3.0 What was found missing as we advanced with UNICOS

Having an established history of enhancing our systems and being blessed with a very able body of knowledgeable system analysts we were in a very good position to identify what was missing (for us) in UNICOS. The following is not in any priority order nor is it a definitive list of missing items, but does capture the more noteworthy ones, some of which are site specific. By ‘missing items’, it is meant those items deemed necessary to operate the system to satisfy the internal data center and Boeing engineering customer requirements. Some items are given with a minimum of descriptive text, but sufficient enough to provide the reader with a good level of understanding. Other ones which the author deemed to be more challenging and/or complex or probably of interest to a wide range of other sites are provided with a lengthy detailed design description covering subject background, implementation issues and, in some cases, further areas for future work or study.

3.1 Integration, Configuration Control & System Building Procedures

These procedures are considered essential for being accountable to ad-hoc audits, simply good business practices, and are needed for being able to reinstate old systems for establishing a computing environment to rerun some past engineering design analysis. This may sound mundane but is considered extremely important to the Boeing Company.

The basic tools for carrying out these tasks are all provided within the UNICOS release software. However, no documentation or cohesive feature for accomplishing them is offered by the vendor, and so a site needs to develop their own. This came as no surprise as the requirements of each customer site undoubtedly differ greatly making it a monumental task or even impossible for the vendor to satisfy. A fundamental requirement for Boeing was to have buildable system source code which was secured very early on in the procurement cycle and when Cray had little reservations concerning the release of source code with respect to their internal maintenance processes. Since then, much has been debated

over the access to system buildable source code with the Cray customers prevailing in the end.

3.2 Human Process Error Prevention

Human beings can at times cause tremendous disruption to the health of a production computing system when they perform well-intentioned but wrong actions or simply unintentional ones. This aspect is especially serious when data center analysts with root privileges perform these type actions while executing in super-user mode. The most alarming things can happen when someone employs cut-and-paste operations on a workstation and clicks the wrong mouse button or clicks it at the wrong time!

The Process Error Prevention (PEP) design for preventing catastrophes focuses on preventing accidents while one is super-user. The single functional requirement is to reduce the number of system interruptions associated with careless system administration activities.

All super-user activities are conducted under the command ***rsu*** which is a local modification to the ***su*** command. The ***rsu*** command associates a user account with a password and is only useful to selected system administration people. The most common mistakes that have been made in the past revolve around:

- /bin/rm usage resulting in removal of critical files
- erroneously updating root cron entries; running multiple cron-s
- modifying root owned configuration files
- modifying file permissions
- executing mis-behaving scripts

Probably the most important enhancement in this design is the change made to /bin/rm. Most users do not specify the -i parameter because it results in a vast number of required responses. Thus, if -i is not specified the /bin/rm command will act as follows when the RM_CONFIRM variable is set (by default, when ***rsu*** is executed RM_CONFIRM variable is set automatically):

- display the current working directory
- display what files and/or directories have been specified
- ask user if they wish to continue, and if ‘no’ exit with exit(0)

Exception to this action takes place if executing *make, nmake, /etc/shutdown,* and the installation Menu System.

For example if a super-user is positioned in $TMPDIR and enters */bin/rm test/** and in this directory there are file names teddy, rufus, and fella then, when executing under ***rsu*** the following displays result:

rm: current working directory = /wrk/jtmp.007384a

rm file arguments: test/teddy test/rufus test/fella
continue with this rm?

Another very useful facility for logging and synchronizing administrative changes is provided with a command script named ***sysedit***. This command assumes one wishes to edit a file and accesses a lock file and then requests the super-user to first specify the ***rsu*** password and then prompts for the file's full path name. The user is requested to enter a description of the intended file modification, and then allows the user to edit the file. After editing the file ***sysedit*** asks if another file operation is desired, and if not will then remove the locking file to allow other system administrators to perform ***sysedit*** work. The file name and description provided along with a time stamp is recorded in the ***sysedit*** log file located in /etc/README. This logging information has proved to be enormously useful over the years. It has helped with many problem resolutions such as when someone quite innocently makes a quick (but erroneous) change and leaves for several days. Typically the support analyst who is called to deal with the resulting system anomaly will first review the last few entries in /etc/README for possible clues.

Obviously, more can be done to prevent accidental damage to the system. The process of dealing with these evolves over time, and by carefully examining each occurrence one can decide whether to incorporate accident avoidance software similar to that described above for /bin/rm.

Many other common mistake items are addressed within the Health Monitoring System design mentioned later.

3.3 Reboot Speedup & Automatic File System Recovery

3.3.1 Reboot Speedup

The released boot process requires numerous human replies which prolongs the boot process unnecessarily as the same replies are required every time. This feature automates the entering of the same replies to help avoid type-in errors and to maximize the service availability.

This feature was first introduced at the UNICOS 6.1 level to eliminate the delays between a system panic and the start of the reboot. The recovery from a system interrupt involves to steps; securing a system dump and rebooting the system. The Reboot Speedup automates both of these steps.

When the system panics a script ~cri/bin/cpupanic is called on the OWS and its screen turns red. This script checks if a panic has occurred in the last 60 minutes. If one has, ***cpupanic*** will not attempt another dump, posts "last panic was N seconds ago", and inhibits an "auto" reboot to avoid thrashing on reboots. If one has not, a dump is taken and "bootsys -F" is called to reboot the system. ***cpupanic*** then checks every 5 minutes to see if the system has reached multi-user mode. If this state has not been reached within 30 minutes

and another panic has not occurred, a message is issued and the OWS screen color is changed to yellow. In this event, operator intervention will be required.

The initial run level that /etc/init will achieve will be level 6. At level 6 the script /etc/init_ask is executed. At the beginning of this script a program is started that does asynchronous reads to the system console and asks the operator to respond.

```
***ATTENTION*** Please respond within 1 minute
                a carriage return and the system will
                go to single user mode. By entering
                nothing, it will go to multi-user
                mode automatically.
Waiting for a response>
```

If no response given within 1 minute, /etc/init_ask calls /etc/autoboot panic to go to multi-user mode, otherwise it calls /etc/init S to go to single user mode. When the system eventually goes to multi-user mode our Health Monitoring System (see Section 3.24) handles automatically the starting of NQS queues and the Tape subsystem initialization.

The system boot scripts are modified to automate the boot process. This automation has a manual override to facilitate special test conditions and for dealing with abnormal conditions such as a ‘system panic’ and for possible problems within the auto-boot process. The system is automatically started with a single input - **auto**.

3.3.2 Automatic File System Recovery

The automatic file system recovery is accomplished by calling a modified /etc/mfsck with the -a option. In addition, the /etc/fsck was modified to return a non zero exit value if it failed for any reason. The modifications to /etc/mfsck were simply to continue forking the /etc/fsck if it returned non zero exit, up to a maximum of ten times. From our experience we have seen that after the first /etc/fsck call (the “read” preening phase) that most file system errors are cleaned up by the fourth pass. This facility is a huge improvement over what was available in earlier UNICOS days.

3.4 System and Software Product Level Selection

From our experiences with the COS system we knew that changing system levels (e.g., going from UNICOS 7 to UNICOS 8) or a product-set (versions of a compiler, libraries, header files, loaders, linkers, and associated man pages are defined as a product-set) can place a burden on system administration staff’s time and energy as well as the user community. To eliminate the headaches associated with changing levels of this kind, a design was required to facilitate a near transparent conversion between levels. Independence between system and product-set levels is considered highly desirable.

For example, the system administrator requires to build UNICOS 8 products while executing under UNICOS 7 (UuU not withstanding) or the user who wishes to continue running

UNICOS 6 product-sets when executing under UNICOS 7. The last example is real as UNICOS 7 product-sets introduced 16 compatibility issues with UNICOS 6.1. These issues were in the areas of a) binary compatibility, b) commands, c) segldr, and d) C compiler.

Users control the product-set level by defining the global environment variable SYSLVL. The default SYSLVL value is defined by the data center. For example, to protect Boeing users from the compatibility issues mentioned above we defined SYSLVL=6 when we installed UNICOS 7. Only after some agreed upon period did we change this to “7”.

To support the SYSLVL feature the front-end scripts for the compilers are changed to handle the environment variable SYSLVL. Also, front-end scripts are create for the assembler (/bin/as), the loader (/bin/segldr and /bin/ld), the assign command (/usr/bin/assign), the asgcmd command (/usr/bin/asgcmd), and the man command (/usr/bin/man). In these front-end scripts, if SYSLVL equals the current OS system level, no command parameter changes will occur - they will merely execute the default system commands. However, if SYSLVL equals, say 7, commands in /usr/lbin_7 will be executed plus command parameters may be changed. For example, /bin/segldr will execute /usr/lbin_7/segldr and command parameter references such as "-l /lib/libc.a" and "-L /usr/lib" will be changed to "-l /lib_7/libc.a" and "-L /usr/lib_7" so that the appropriate libraries will be used.

There were two limitations apparent in this method when upgrading to UNICOS 8 and user defining SYSLVL=7:

1. If full-path names are specified for header file inclusion in C source code, they will not be detected. For example, "#include </usr/include/errno.h>" will not be detected and the user will get the UNICOS 8.0 version of errno.h. If users have code employing full-path names such as this, the ramifications are that probably the compilation or link phase will fail.
2. If any file names containing ***segldr*** directives are specified in the environment variable SEGDIR, they will not be scanned and temporarily changed. Thus, references such as LIBDIR=dir in directive files may cause UNICOS 8.0 libraries to be referenced.

Over the years when faced with upgrading the major UNICOS OS system levels this feature has allowed Boeing to almost transparently move users to the new levels. This implied transparency allows the data center to adopt an upgrade schedule that is less likely to be undermined by users who, at the last minute, decide the new product-sets are causing problems for them or that they are being forced to recompile and reload, and because of this require more time to certify their codes for production use.

Obviously, this scheme requires the site to provide additional disk space to accommodate the multiple system level product-sets. Typically, we provide for no more than two levels, although this is not a restriction. For example, when we install UNICOS 10 the UNICOS 9 product-sets will require some 270 Megabytes of disk space. I might add here, that this space also accommodates the libraries needed for UNICOS 9 C90 and TS modes as well as those to support Programming Environment 1 (Programming Environment 3 is our default under UNICOS 9 and 10).

3.5 Remote OWS

We have provided our data center operators with a remote OWS. This remote OWS is located several hundred feet away from the T916 and is located in the data center's command center room. This room has system consoles for all of our computing platforms. The remote OWS is a SUN workstation connected to the real OWS by a local ethernet.

To switch control from the real OWS to the remote OWS the command ***rows*** is entered in any window on the real OWS. This causes a set of five SYSOPS windows to appear on the remote OWS. While switched to the remote OWS, any screen color changes occur on both OWSs. Thus, if a bootsys is done in the "(CONSOLE)" window of the remote OWS the root screen color for the remote and real OWS will change to what is specified by BACKGROUND in the /etc/configfile. If a system crash occurs while switched to the remote OWS, a new console window will appear on the remote and the root screen colors will go from red to gold to green.

To switch from the remote to the real OWS the command ***ows*** is entered on the remote OWS.

3.6 Disk space Reservation

The disk space reservation scheme is simple in design and relies on DMF for providing the mechanism to free disk space. Although there are some weaknesses in the design, it does satisfy 99% of the overall requirements for avoiding job aborts due to lack of disk space. Over the last six years this scheme has proven itself to be both robust and reliable, low maintenance, and useful to Boeing.

NQS jobs which use large amounts of disk space have to be single-streamed through a special queue. The disk space reservation scheme allows a portion of a DMF-managed file system to be set aside as reservable. Thus, by allowing NQS jobs to specify how much disk space in the file system is needed via a special command, multiple NQS jobs can be run simultaneously so long as their total reserved space can be accommodated. Each job will run in its normal queue, removing the need for a special queue.

The following functional requirements are provided:

1. Users must have the capability of reserving file system space
2. The capability to reserve space should be easily extended to multiple file systems
3. Users and system administrators must have visibility of current guaranteed space reservations
4. Space reservations must be retained across scheduled and unscheduled system interruptions for recovered jobs. Conversely, for jobs not recovered, space reservations must be released
5. The disk space reservation activity must be time-stamped and logged. All delays in granting the space reservation should be tracked and be available for reporting

6. The pool of file system space that can be reserved must be adjustable at all times and should not be limited to the physical size of the file system to allow over- and under-subscription

The bulk of the work is performed by a daemon script (i.e., no code) which is launched at boot time. This daemon performs all of the tasks required to deliver guaranteed disk space to a requesting job as well as the recovery activities needed immediately following a boot. Granting a space reservation consists of making sure that the requested amount fits within the file system limit for reserved space and then issuing a dmfree command if there is not enough free space on the file system.

To request reserved space, a job executes the command /local/bin/rsp. The job is held at this command until the request is granted. This command is not honored unless the job is under NQS control.

The syntax for /local/bin/rsp is as follows:

/local/bin/rsp [-f fsys] space

where *fsys* is the name of the file system and *space* is the amount of disk space being requested for reservation. The *fsys* argument defaults to /big (site specific). The *space* argument can be just a number which will be interpreted as bytes or can have suffixes of Kb, Mb, or Gb.

The space reservation mechanism will be run on the /big file system only (site specific requirement). One half of this file system will be set aside for reservation and a single request will be limited to this also.

All files dealing with space reservation reside in the directory /usr/spool/rsm (Reserved Space Manager). A /usr/spool/rsm/granted contains a record of current space reservations. All activity will be logged in /usr/spool/rsm/rsmlog. The amount of space subject to reservation will be kept in /usr/spool/rsm/rsm.conf. The daemon will read this file on every pass in case it has been updated.

The reserved space manager daemon, ***rsmdaemon***, will perform the following tasks:

1. Reconcile /usr/spool/rsm/granted with the list of recovered jobs at system boot time. Jobs which are not recovered will have their reservations removed from the file.
2. Read reserved space requests from directory /usr/spool/rsm/requests. The daemon will grant the request immediately if possible. If the requested space will fit within the limit on reserved space but there is insufficient free space on the file system, the daemon will issue a ***dmfree*** command to release enough disk space. If the requested space will not fit within the limit on reserved space, no action will be taken. In either of the latter two cases, the daemon will issue a ***qmgr*** hold command on the requesting job to remove it from execution. When the request can be granted, the job will be released using the ***qmgr*** release command and the request removed from /usr/spool/rsm/requests.

3. Read /usr/spool/rsm/rsm.conf for changes in the amount of space allowed to be reserved
4. Maintain a time-stamped log of all events in /usr/spool/rsm/rsmlog
5. Monitor the free space on the file system. If it falls below a specified threshold, all jobs with space reservations will be held (using the ***qmgr*** hold command) until the free space rises to a sufficient level. These two thresholds will be initially set to 4% and 8% of the file system respectively
6. Monitor running jobs via ***qstat*** commands. When a job with space reservation terminates, /usr/spool/rsm/granted will be updated

The command executed by a job to request reserved space, /local/bin/rsp, is a simple script which performs the following steps:

1. Record the request in /usr/spool/rsm/rsmlog
2. Write the request into a file in /usr/spool/rsm/requests
3. Go into a sleep loop waiting for the request to be removed
4. Record that the request has been granted in /usr/spool/rsm/rsmlog

There are two main reasons for concentrating so much of the work in the daemon and keeping the user command simple:

1. Several of the steps that may be needed to grant the space reservation require root or ***qmgr*** privilege. This design allows the user command to remain unprivileged.
2. Timing problems can be avoided. The /usr/spool/rsm/granted file is maintained by the ***rsmdaemon*** alone. The /usr/spool/rsm/rsmlog will be used by both ***rsmdaemon*** and the user command ***rsp***, but only in write mode.

When ***rsmdaemon*** determines that a ***dmfree*** command is required, it will compute the target for free space as follows:

1. A 2 Gbyte pad needs to be maintained (largest file size limit allowed - site specific) so that a user can ***dmget*** a file of the largest allowed size.
2. If there are no current reservations, the target free space is simply the reservation plus the pad.
3. If there are one or more current reservations, the target free space is the current free space plus the new reservation.

Implicit in this definition is the fact that a space reservation is only guaranteed at the moment it is granted. There is no feasible way to keep other processes from using the space if they want.

The problems and/or benefits provided by this feature are:

- The scheme described in this design is restricted to use in NQS batch jobs.

- The scheme works on a strictly first come, first served basis. No attention is paid to priority or turnaround level in granting reservations.
- Reserved space, once granted, cannot be guaranteed for the life of the job. There is no way to earmark a portion of a file system for use by only one job.
- Disk space can be reserved on /big file system only (site specific - this file system has a very large average file size blocks to inode ratio; 200 to 300). The scheme could be extended to other DMF-managed file systems without difficulty, but vastly smaller average file size blocks to inode ratios (of say, less than 10) would likely make it work much less well.
- It is readily apparent that the more users of /big (site specific file system) that utilize this scheme there are, the better it will work. The more processes there are using unreserved space, the more likely it is that reserved space will not be available for the full life of a requesting job.

3.7 Multiple System Dump Areas

Provides the capability for having two system dump areas on disk so that to the extent possible a system dump can be obtained if one of the dump areas is inaccessible. Obtaining system dumps is deemed important as the vendor cannot or at least will find it very difficult to solve a system problem without them.

The OWS cycles between the two dump areas when a system dump is taken. This technique ensure that both areas are exercised so that the neither lays dormant. Each dump area is a replica of the other but is located on a unique disk and i/o path each using different IOPs.

When the system is booted and eventually goes to multi-user mode (i.e., /etc/init 2), the script /etc/brc is called which in turn calls /etc/coredd to check if a system dump is present. The /etc/coredd will check one dump area first and if a dump is present the area will be assumed to be a dump image. If a dump is not found in the first area checked then /etc/coredd will examine the other dump area for a dump image. After /etc/coredd completes, /etc/bc initializes both dump areas.

The following components are modified:

/etc/configfile and ~cri/bin/mfdump on the OWS. The latter component is a front-end to the actual mfdump which is renamed to .mfdump. /etc/configfile is enhanced to contain a set of environment variables describing the two dump areas and the last dump area used.

/etc/brc modified to initialize both dump areas.

/etc/coredd modified to located unused dump area to use.

/etc/cpdmp modified to accept its last argument parameter as a dump area. If this parameter is omitted /dev/pdd/dump is used otherwise /dev/pdd/dumpB is specified.

3.8 Job Submission via ftp

We developed a job submittal enhancement to ***ftp*** at the UNICOS 5.1 level to service job submission requests coming in from non-UNICOS computer users.

This feature allows users to submit NQS jobs to UNICOS via ftp. This was a requirement for our users of CDC (Cybers), IBM Personal Computers (PCs) and Apple Macintoshes (Macs) which do not normally run the Remote Queueing System (RQS) software. The only common software running on these systems was ***ftp***, so it was used as the baseline to support the job submission function. It was a requirement to avoid making changes to the ftp software at the PC and Mac workstation.

All code modifications in support of this project was done on the Cray side. It was intended that this code would be usable from a wide variety of workstations, the only requirement being some implementation of ftp exist. However, this code would not be a replacement for RQS. RQS (or the locally developed RNQS which, incidentally now competes with the NQE product from SGI/Cray) is preferable for those workstations that support it since it provides many more features, including automatic return of output.

The ftp job submission described below was intended for those workstations that cannot run RQS.

The functional requirements were:

- To support the submission of batch jobs to the Cray NQS system via ***ftp*** (***qsub***)
- To support the return of job status on jobs submitted (***qstat***)
- To support the control of jobs submitted (***qdel***)

The ***ftpd*** server on UNICOS was modified to recognize a job submission request by checking the remote file name on a put command. If a job submission was detected, it would fork a ***qsub*** process and divert the file transferred to ***qsub*** as its standard input. Command line options for the ***qsub. qstat, and qdel*** commands were supported by using colons as the delimiter. This avoided the problem that some ftp clients did not support blanks embedded in file names. The request-id of the submitted job was returned to the client after the transfer was complete.

With UNICOS 8 the facility for submitting NQS jobs from ***ftp*** was provided. We however, have kept our ***ftp*** job submittal intact for continuity reasons and because it offers slightly different capabilities.

We would very much like to see either the Boeing or Cray ***ftp*** NQS job submit feature in the IRIX system, and encourage other member sites to so state as well. For this reason the following implementation details are provided.

3.8.1 The *ftp* get and put commands

The batch commands take the general format of either the ftp put command (for job request submission) or the ftp get command (for gaining status or controlling (e.g. deleting) a job).

These formats are:

```
put local-file remote-file
get remote-file local-file
```

When the ftp get and put commands are used for UNICOS remote batch processing, the user substitutes for the remote-file name one of three commands:

qsub: for submitting a job request
qstat: for requesting status
qdel: for deleting or signalling a running job

Each option and each option value for these commands follow immediately after (i.e. no blank spaces). And each option or option value is delimited by a colon.(Colons are used instead of spaces as delimiters because some ftp implementations will not handle spaces.)

Below is an example of the ftp command for receiving job status:

```
get qstat:-r:-l my_file
```

Spaces separate the ftp command, the remote-file name (qstat:-r:-l) and the local-file name (my_file). In the remote-file name itself, colons, rather than spaces, delimit the options (and any option values).

The ***qsub, qstat*** and ***qdel*** commands and their respective options are part of the UNICOS Network Queueing System (NQS) which handles the requests for batch processing and receives the commands from the ftp software.

When the user ftp-s to UNICOS, the same environment variables apply as if you were to log in to UNICOS and submit jobs directly with NQS (i.e. QSUB_ variables are defined with ftp just as they would be when qsub is invoked from the command line).

3.8.2 Submitting a batch request (qsub:)

The NQS command qsub allows batch job submission. The ftp put command is used to provide the interface to this qsub command on UNICOS.

The put command followed by the name of the batch-request script file to causes the script file to be submitted. This is the local-file. Most implementations of ftp will also

allow a full file path to the script file. The script file must be in NQS batch-request script file format.

The script file name should be followed by a space and the qsub: command (include the colon). The example below shows the basic command format:

```
put script qsub:
```

where script is the name of the batch-request script file.

In addition to the basic form above, qsub: will accept all valid command options of NQS qsub. Any option follows immediately after the command, delimited by a colon. Additional options and option values are likewise set off by colons rather than spaces. For example:

```
put script qsub:-q:ME:-lT:600:lM:10
```

will transmit file script to UNICOS. The :-q:ME option submits it to NQS in medium service class (ME). The :-lT:600 option (and value) sets a CPU time limit of 600 seconds. The :lM:10 option (and value) sets a memory usage limit of 10 megawords.

Successful submission of a job request will result in an NQS request identification number being returned to the user. This number can be used to display status of the job (qstat) or delete it when processing (qdel) (see below).

Note: NQS qsub options are also available for use when entered in the script file.

3.8.3 Requesting status (qstat:)

The NQS command qstat displays the status of the NQS queues and job requests. The ftp get command is used to provide the interface to qstat.

The get command should be followed by a space and the qstat: command. Follow qstat: with any of the available options. Delimit options by a colon. Note: if no options are used, you still must end the qstat command with a colon for proper interpretation by the UNICOS software.

The qstat command and any options should be followed by a space and the name of the file to receive the status information. The file is be placed on the user's workstation.

The basic batch status command is shown below:

```
get qstat: myfile
```

where myfile is the name of the file to receive the status. If no file name is specified, ftp will create a file with the name qstat:. Thus, user should always provide a file name for the status.

As an alternative to the file name for status information, a - character can be used. In most implementations of ftp, the - character sends the status information to the screen rather than a file.

Below is an example of an ftp status command with the information sent to the screen:

```
get qstat:-r:-l -
```

where -r is the option to view currently running requests in pipe and batch queues, -l is the option to view the summary of queue limits, and the - character is the request to send the information to the screen.

3.8.4 Deleting/Signalling a running request (qdel:)

The NQS command qdel deletes or signals running batch requests. The ftp get command provides the interface to qdel.

The get command should be followed by a space and the qdel: command and, immediately after, the batch request identifier (returned when you submitted your job with qsub). After another space, specify a name, to send qdel output messages to a file, or type a - character, to send output to your screen. Example:

```
get qdel:12321 -
```

The reason qdel is implemented with the get command is that get provides the mechanism to return the standard output of the command (e.g. notification that the job has been successfully deleted) as a file transfer to the workstation user.

Any of the NQS options can be used with qdel (see below). As with qsub and qstat options, qdel options immediately follow the command and are delimited by colons.

An example of qdel, using the -k, kill option, to kill the running batch request 12321 is:

```
get qdel:-k:12321 -
```

where the - character specifies that the output messages be sent to the user's terminal.

3.9 Catastrophic Disk Failure Recovery

The disk subsystem consists of many individual disk devices. When one of these disks fails, it is necessary to repair the unit and then possible rebuild and reload file data. The rebuilding and reloading of data is complicated by the fact that each disk contains many file systems, and file systems may span disk units. Backup data is dumped by file system, not by disk unit. Therefore the reloading of a disk unit can be a complicated process of identifying, rebuilding, and reloading many file systems across multiple disk units. This feature implements a set of semi-automated procedures which allows extremely rapid and foolproof reloading of a disk in the event of a failure; the only human intervention being the specification of the failed disk unit(s).

Some special actions will be required in the event that the DMF data base is lost, or the dump/reload data base is lost, or the root device is lost.

This feature has been used infrequently, but when it is, it is an invaluable asset.

3.10 Data Migration Facility (DMF) Enhancements

Over the years working with DMF we have developed a number of local enhancements. Today we configure the client/server DMF 2.5.5 under UNICOS 9.0.2.8 which is managing 20 Terabytes of data and 2.3 million files. The average annual data growth is 6.25 Terabytes, so by mid 2001 we expect to be managing some 39 Terabytes of data. The majority of our local configuration mods are made to the DMF configuration file. The configuration file in use depends on whether we are running the UNICOS guest or running the client/server on the host system. The more significant local mods are given below.

- ***dmdstop* flush code**: A"-f" parameter was added to ***dmdstop*** to flush the msps and allow the ***dmdaemon*** to finish its active requests (this allows dmget-s to be gracefully terminated during system shutdown). After flushing, the ***dmdaemon*** waits 2 minutes to allow the EOT descriptors to be written on tapes.
- ***dmget*-s delayed due to low disk space:** The dmf_config parameters controlling this are:

  FS_MIN_FREE - Minimum free % before dmget-s are delayed - default is 5%.

  FS_DELAY_TIME - Secs between rechecking the % of free space for delayed dmgets - default 120

  MAX_DMGET_DELAY - Max secs to delay a dmget request - default 1800

- **DMF express pipe:** An "express pipe" /usr/dm/dmd.xreq.pipe is added to service ***dmstop*** and ***dmdstat*** requests. I hope the need for this is obvious to the reader. If not, a brief description for this is as follows. Our Health Monitoring System (see Section 3.24) periodically attempts to verify that DMF is "alive and well". It uses ***dmdstat*** for this purpose. If DMF is busy servicing requests the ***dmdstat*** can take considerable time to be noticed by the ***dmdaemon***. The "express pipe" helps avoids false alarms from the Health Monitoring System. The same is true for ***dmdstop*** which is used during scheduled system shutdowns.
- ***dmmigall* -s parameter:** A "-s" parameter was added to specify a minimum file size for migrating files (N.B. A bug was fixed to implement the -l parameter).
- **Requeueing secondary msp requests:** We added code to retry the primary msp copy once, after a dmget from the secondary msp copy failed. This was to avoid trips (a 4 hour retrieval time) to the offsite tape vault when the primary msp tape copy is marked "hbadmnt" for no good reason.

One other local enhancement of note is the ***mvhandle*** and ***cphandle*** commands (which I believe were authored by GE originally but we received it from Cray in February 1994 as an unsupported feature). The ***mvhandle*** command moves a migrated file(s) to another filesystem without unmigrating it while ***cphandle*** will copy a migrated file(s) to another filesystem, and unmigrating them in the destination filesystem.

The command syntax:

mvhandle [-f] files target

cphandle [-f] files target

The primary function of the ***mvhandle*** utility is to move migrated files to another file system without unmigrating them unlike ***mv*** which would. The usefulness of ***mvhandle*** is in moving large migrated files to another file system. Since the file is not unmigrated to do the move, the source file system as well as the destination file system is not impacted by the move. Disk space does not have to be made available since the file stays migrated. If a file is dual-state, the file, when moved to another file system, will not maintain its online copy. It will lose its dual-state status and only be off-line. The dropping of the online copy is done for performance reasons. Moving a migrated file is very quick compared to copying a large online file. ***mvhandle*** can also be used to move migrated as well as unmigrated files within a file system, but this is not ***mvhandle***'s strong point. Moving files within a file system is just a rename system call. Also, ***mvhandle*** can be used to move unmigrated files to another file system, but this operation is just a copy.

The primary function of ***cphandle*** is to copy migrated files to another file system without unmigrating them in the source file system. The file is unmigrated in the destination file system to make the copy. The copy is a regular file with no connection to the source file. The source file remains migrated and the destination is merely a copy. The usefulness of ***cphandle*** is in copying migrated files to another file system since the file is unmigrated in the destination file system and not the source file system. Disk space needs to be made available in the destination file system, but not the source file system. If a file is dual-state, the online copy will be used in making the copy. The file will not be unmigrated in the destination file system in order to make the copy. ***cphandle*** can also be used to copy migrated files within a file system. ***cphandle*** operates in the same manner as when copying files to another file system. Since the source and destination file systems are the same, the file will be unmigrated in the source file system. The only advantage to using ***cphandle*** for this case is that it can be done with one command instead of "dmget file ; cp file target" and the source stays migrated.

cphandle can also be used to copy unmigrated files within a file system or to another file system, but this is not ***cphandle***'s strong point. Copying files in this case is merely a copy.

Boeing would very much like to have all these DMF enhancements ported to IRIX.

3.11 Tape Volume Management System (TMS)

When we first installed UNICOS 5 there was no feature that provided management for tape volumes. Since then, Cray has released the Reel Librarian and subsequently the Volume Management Facility. However, these feature releases came too late for Boeing as we needed the feature in 1989. For this reason we developed our own (basically we ported what we used on the COS 1.14 system). Tape volume management is a critical need for

data center management as well as the end user. In fact, the data center is by far the biggest user of our TMS.

It should be noted that with UNICOS 10 the UNICOS Tape Subsystem is released as a separate product in binary form. A new feature with this is the introduction of “user exits”. As a result of this, all of our local tape subsystem modifications have been redone to use these new user exits. The implementation details for this are provided in the following Section.

The commands we have developed for TMS are as follows:

tmsrsv - reserve a tape volume
tmsls - list volume names
tmsrm - release volume ownership
tmsxfer - transfer volume ownership
tmsmod - modify volume information

where

tmsrsv [-g dev-grp-name] [-l labeltype] [-x expir] [-c comment] [-G group] [-a acct] [-m mode]
tmsls [-F] [-L] [-v vsnlist] [-V vsnfile]
tmsrm -v vsn [-V vsnfile]
tmsxfer -v vsn [-V vsnfile] -u userid
tmsmod -v vsn [-V vsnfile] [-x expir] [-c comment] [-G group] [-a account] [-m mode]

These are utilities to allocate, list, delete, transfer ownership, and modify attributes of allocated scratch tapes. The bracketed items are optional parameters.

Options:

-g device-group-name

Specifies the name of the device group. Either "CART" or "TAPE", with "CART" being the default. "CART" is the preferred media, since it holds more data, is more error free, and is faster.

-c comment

Comment line (less than 80 characters)

-l labeltype

Type of label: "al" ansi-labelled, "sl" IBM-standard label, "nl" not labelled. "al" is the default and is preferred.

-v vsn

Vsn or vsn(s) separated by colons (:).

-V vsnfile

A name of a text file that contains vsn(s). -V and -v are mutually exclusive.

-u userid

A userid to transfer tape ownership.

-F

Generate a "full" list. If not specified, tmsls lists just vsns. If -F is specified, the following is written to the standard output: vsn, userid, device-group, label-type, creation date, expiration date, and comments.

-L

Generate a "long" list. If not specified, tmsls lists just vsns. A "long" list contains items listed with the full list, plus additional ones.

-x expir

This parameter specifies the days left until the volume expires. This is different from the label expiration date as specified by the tpmnt command. If expir is 0, then the tape is held indefinitely. This is the default.

-G group

This parameter specifies group membership for the tape. There is only one group associated with each volume. The default for tmsrsv is the group id under which the process runs.

-a account

Specifies a billing account to bill tape ownership. The default is the account under which the process runs.

-m mode

An octal number specifying who may access the volume.

Bit 0400 read by owner
Bit 0200 write by owner
Bit 0040 read by group
Bit 0020 write by group
Bit 0004 read by others
Bit 0002 write by others

No other bits are defined. The default is -m 0640.

Five additional commands are provided for data center personnel, and just a brief description is provided here.

tmsdb -ADRSELINQ -U upfile -p poolid -l label -g generic_type -s status -u userid -v vsnlist -V vsnfile -c comment -x expir -w location -r rackid -G group -a account -m mode -o prefix -n prefix

tmsrsv -g generic_type -l label -x expir -c com -G group -a acct -m mode -p poolid -u userid -S silostat

tmsls -RW -p poolid -s status -u userid -G group -a account -S silostat

tmsrm -v vsnlist -V vsnfile -u userid

tmsmod -v vsnlist -V vsnfile -x expir -c comment -G group -a account -u userid

These commands provide: add volumes, delete volumes and revise information about volumes in the tms database. To execute these program requires the user to be super-user, taplib, or a member of the operator group.

In addition the utilities ***tmsrsv, tmsls, tmsrm*** and ***tmsmod*** allows the operator to perform their function on behalf of the user specified by -u userid. ***tmsrsv*** also allows the operator to allocate from any specified pool or silo domain, and ***tmsls*** allows the operator to list vsns in a specified pool, or with any specified status, group, account, or silo domain (in or out). These additional functions are not given to the user.

Boeing intends to port TMS to IRIX.

3.12 Tape Management Facility (TMF)

As mentioned above, the UNICOS 10 system release has caused us to re-think our local code modifications to the UNICOS TMF. Use of user exits is made as follows. These are mentioned in the UNICOS 10 Release Overview notes (RO-5000 10) and fully documented in the UNICOS Tape Subsystem Admin Guide (SG-2307) which is online at:

http://swpubs-internal.cray.com:8085/library/all/2307_10.0/1604

For our purposes the modules /usr/src/cmd/c1/tp/tpuex.c and /usr/src/cmd/c1/tp/tpuex.h are modified. The recoding effort was actually very straight forward, once we understood how the ‘exits’ worked. We took the whole of our TMS module and locally inserted it at the end of tpuex.c and modified other tpuex.c areas to make calls to this module. This module name is tms() and is given in more detail below:

```
tms(irw, req)
int irw;              /* read=1, write=2, process_scratch=3 */
struct uex_table  *req;
{
   struct tmsvaxrq {       /* Validate Volume access request*/
      int magic;      /* magic number */
      int uid;              /* requestors uid */
      int gid;        /* requestors gid */
```

```
    int rwflag;     /* read=1, write=2 */
    char vsn[8];        /* vsn */
    int scr;        /* scratch request flag */
    char pvsn[8];       /* previous vsn, if any */
    int volume_sequ;    /* volume number in set */
    int acctid;     /* requestors account id */
    int label;      /* label type */
    long int density;   /* density */
    long int mbl;       /* block size */
    long int lrecl;     /* logical record size (if any) */
    char recfm;     /* record format */
    char device_id[16]; /* device name */
    int res;        /* reserved for expansion */
} req_tms;
```

This single module allows us to validate tape volume read and write requests and for processing additional tape volumes for file system dumps when ***dump*** exhausts its given VSN set. When ***dump*** exhausts its VSN list it is necessary to obtain another tape volume from the appropriate scratch tape pool and assign it to the ***dump*** process and its VSN set. This aspect assures us that the file system dumps will never abort due to insufficient VSNs being specified, as is the case with the released ***dump*** command.

Todate, we have only been able to test this recoding effort under the UNICOS guest system running UNICOS 10. So far, testing has shown it to work as designed.

The “user exits” provided along with the binary form have allowed us to continue applying our local modifications to UNICOS TMF.

The UNICOS TMF feature together with its ‘user exits’ should be included in IRIX.

3.13 Data Security

Data security addresses concerns related to file access controls, system penetration, and classified processing. A high degree of data security is provided by default in the UNICOS system and prevents unauthorized customers from obtaining any information which can be used to penetrate other customer accounts; prevents customers from finding out about other users of the system outside their group; provides an optional reporting facility detailing security related activities and prevents access to system resources other than for authorized purposes. Elements of the data security feature protect certain sensitive files such as the password file from being read by users, restrict use of the super-user account, modify system commands such as ***who*** and ***ps*** so that information about other users is restricted, modify the ***mail*** command to strip out non-printing characters, establish more restrictive defaults for file access by users other than the owner, and remove the capability to disable process accounting.

Utilities for encrypting and decrypting data are available to provide an additional layer of data security for the more sensitive customer data.

The UNICOS system may be run in stand-alone mode to accommodate classified processing. The Department of Defense (DoD) processing requirements, such as clearing memory and disk storage before and after processing, are satisfied.

The functional requirements we developed for Data Security were as follows.

- Prevent unauthorized customers from obtaining any information which can be used to penetrate other customer accounts.
- Provide visibility of all security related activities; i.e., record failed attempts to log on or to change user id.
- Prevent access to system resources other than for authorized purposes.
- Minimize a user's ability to observe other activity on the system.
- Minimize a user's ability to compromise or adversely affect another user's interactive logon.
- Enforce password expiration to minimize damage resulting from an account break-in.

Some of the above requirements need further explanation.

Preserving the integrity of user and system data must be the fundamental objective in any security scheme. Due to the collegial nature of the UNIX system and its development, this aspect has not received the attention necessary to use the system at a place like The Boeing Company.

Many system support files that we would consider sensitive in nature are readable by everyone, the idea being that, with very few obvious exceptions, just reading a file cannot constitute a security breach. This aspect needs to be judiciously restricted. Further, any modifications that increase the difficulty of penetrating the system from the outside should be investigated.

It has been the practice at Boeing to consider the identity of our clientele to be privileged information. This being the case, we cannot allow logged-on users access to information about who else is using the system or about what other users are doing. Several commands are designed for just this purpose and need to be restricted.

In the unlikely event that the system is penetrated, logs of security-related activities will be needed in the search for the perpetrator. It generally takes many attempts to guess a user's password and evidence of these failed attempts should be available as evidence of when and from where the penetration occurred.

On most UNIX systems, the user has the ability to write to another user's terminal, either directly or through the mail utility. A malicious user could send a message with embedded escape sequences which could cause havoc to the recipient's session. A sophisticated user

could even devise a scheme which would endanger other users' files. Some way to limit a user's exposure to this danger should be implemented.

Changing passwords on a frequent basis makes breaking in to a system all the more difficult. As many user's as possible should be required to do this as a matter of common sense.

Significant enhancements in the security area were added to UNICOS 4.0 release. Some of the new features were access control lists for individual files, a system of security levels and compartments assigned to users and files to restrict data interchange between users, enhanced logging of security related incidents, and a set of privileges which can be granted on a user by user basis. Although UNICOS security is designed as an all or nothing package, some of its features can be left unused even if they are available. In this manner, the desirable portions can be utilized without undue impact to the user.

We decided very early on that the UNICOS security feature should be turned on. This provided access control lists to individual files, enhance security logging, and restricted the use of SUID and SGID programs. At least initially, no compartments would be used for user files and all users would be given the same security level. This made most of UNICOS security invisible to the user. Some use of compartments was however necessary for system administration.

The following files were decided to not be readable by the user:

/etc/passwd
/etc/group
/etc/utmp
/etc/wtmp
/etc/udb.public
/dev/mem
/dev/kmem

The password, group, and udb files contain user id's and other information about everyone allowed to use the system. The utmp and wtmp files contain data about all the users currently logged on to the system. /dev/mem and /dev/kmem are system and kernel memory, respectively.

It will not be possible to log in as the superuser or to ***su*** to the superuser account except at the system console. Special exception to this was provided to a set of internal users however (see below and the Section 3.2 on **Human Process Error Prevention**).

Execution of the ***who, ps, last, udbsee, jstat, groups*** and ***finger*** commands will show information only about the user and the members of his group rather than about all logged-on users.

The ***mail*** command will be modified to eliminate all non-printable characters before displaying messages on the screen.

The default ***umask*** in /etc/profile will be changed to 077 and the command ***mesg -n*** will be added.

In more detail this meant:

- Files /etc/passwd, /etc/group, /etc/utmp, /etc/wtmp, /dev/mem, /etc/udb.public and /dev/kmem will have their modes changed to remove all access by others. They will also have their group identity changed to a new group introduced for this purpose. All programs that read or write these files will be changed to SGID to this new group. Data center personnel that need access to these files can be made members of this group.
- The usage of ***su*** to change to the super-user will be restricted to a small set of data center personnel (see the Section 3.2 on **Human Process Error Prevention**).
- The modules ***who, ps, finger, jstat, udbsee, last***, and ***groups*** will be modified to test whether the user executing the command is a member of the special group owning the files listed above. If not, only those users or processes belonging to the user or his group will be displayed. If yes, the commands will work as normal and display all information.
- The ***mail*** command will be modified to examine all messages and strip out all non-printing characters except for bell, backspace, horizontal tab, line feed, vertical tab, form feed, and carriage return.
- The file /etc/profile which is executed at login for all users will be modified to add the commands ***umask 077*** and ***mesg n***

The above meant that no external changes were needed to existing commands and no new commands need be introduced. The differences the users will notice would be their inability to read certain files, the lack of information about users outside their group when the ***who, ps***, or ***finger*** commands are executed, and the more restrictive ***umask*** default.

The Problems and/or Benefits we saw with this data security design were as follows.

Standard UNIX has some security features related to user passwords that are of interest. Passwords have to be at least six characters long, must have at least two letters and one digit or special character, and must not be a circular permutation of the logon id. In addition, accounts can be set up so that passwords must be changed after a specified maximum interval and/or cannot be changed in less than a specified minimum interval. This last item is to prevent a user changing a password at the required time and then immediately changing it back to what is was before. The ***su*** utility for changing user ids keeps a log of all its activity, whether successful or not. Failures to provide the correct password at initial logon in three tries are logged at the system console.

The Boeing Computing Security Requirements Manual was examined and it is our contention that secure UNICOS with the proposed enhancements met its requirements.

It is understood that the above mentioned access restrictions on usually public files may cause some of the UNIX System V conformance tests not to run. This is regrettable, but the added system security seems to make these changes worthwhile.

Some impacts to our customers who were use to the security on other systems such as CDC/NOS Cybers and the Cray X-MP/COS were as follows.

- There would be no individual file passwords.
- It would be possible, through the ***umask*** and ***chmod*** commands, for a user to open up his files to greater damage than he may be used to. If any directories are made writable by others, all of its files can be deleted, overwritten, or otherwise damaged without the user's knowledge. Care must be taken to instruct users on how to ensure data security.

Boeing is contractually required to allow non Boeing customers (known as suppliers) to use the system. This feature, in part, ensures the integrity of the Boeing data and connected computing platforms while allowing suppliers access the UNICOS system. Even though a Boeing user marks a file as public in the normal UNIX sense, it must be secure from inspection or alteration by a supplier. A contradictory requirement is that a Boeing user must be able to make individual files accessible to a supplier on an occasional basis.

Segregation between Boeing and supplier files was provided as follows:

- All Boeing users are added to a group called 'boeing'.
- All Boeing users have home directories in /u/ba, /u1/ba, /u2/ba, etc. The modes on these directories is 750 and the group ownership defined as 'boeing'. This prevents any user not a member of group 'boeing' from accessing any file in the /u/ba directory tree, for example.
- Home directories for supplier users are setup under directories /u/va, /u1/va, etc. The /u/va-type directory can be changed to mode 750, owned by the appropriate supplier group, to prevent alternate access, at the discretion of the supplier.
- Execution under UNICOS of network commands such as ***ftp*** and remote commands such as ***rcp*** and ***remsh/rsh*** are restricted to members of group 'boeing'. This prevents supplier use of the UNICOS system as a gateway to the Boeing internet. File transfers can still be initiated on a remote host at the other end of the transfer, i.e., network access into the UNICOS system is not being restricted. We restrict entries in the network routing table so that the supplier groups cannot use the ***ftp*** and remote commands. The -gid flag is a Cray enhancement to the ***route*** command.
- File sharing is achieved by copying a file to UFS (Boeing's UNIX File Server), to /tmp or /usr/tmp, or to /u/share (a directory intended to be more permanent than /tmp). No other capability exists to make a file accessible to another user segment without moving it.

The best approach to security is to think about it seriously in terms of what it means to your business. With this in mind we have daily processes that are executed. Some of these are listed below.

- Run the UNICOS facility ***spcheck*** with the default options. Save the outputs each day and use ***diff***s to look for changes. Locally we have added the privilege to create SUID and SGID files to ***spcheck***'s list of non-standard privileges and to vary the periods without any usage that defines a stale account.

- Run ***reduce*** with -t logn to scan the security logs for all login attempts made since the last run. Then grep for errors. Things to look for are login attempts by data center personnel from remote hosts that you do not recognize, login attempts from data center workstations by users that are not recognize, and a pattern of many login failures on different accounts originating from a single remote host (We have only spotted such an occurrence once; it was a real breakin attempt.)
- Scan the su log. This is a standard procedure recommended in all books on Unix security. However, since UNICOS disables an account that has three su failures in a minute and spits out a nasty warning after two failures, there is a strong disincentive for users to try to guess the root or other passwords with su. We have never found any evidence of suspicious activity in our su logs.
- Run the UNICOS facility ***spfilck***. This is another program for checking the integrity of our installed software. It uses a template which lists the owner, group, mode and security label for as many programs as you wish to include. our template includes all the executable binaries we have that are suid or sgid or that have a security label.
- Run the program ***tripwire*** which was developed by Gene Kim and Gene Spafford at Purdue University. This program checks the ownership, group, and mode of a given list of files. It also checks the digital signature of each file, which is sort of like a more sophisticated checksum, but is much harder to spoof than a checksum. our tripwire template has about 1400 files listed in it.

As you can see, there is a great deal of overlap in what these programs do. Our HMS (See Section 3.24) is also doing some of the same sort of checking. However, they each do things a little differently and look at a different set of files.

Tripwire, of course, knows nothing about MLS and does not look at security labels.

In addition to the above, we have a cron entry which disables all accounts that have not been used for 90 days. This is strongly suggested if not required by Boeing security standards. The cron job runs once a week.

On a very infrequent basis (less than once a year), we look for accounts that have not been used for even longer and should be eliminated. This is always hard to do, since you have to figure out what to do with the files.

We have made additions to the passwd command to restrict acceptable passwords. UNICOS comes with fairly stringent limits on acceptable passwords: they must be at least six characters long, contain one non-letter, and the user name or rotations of it are not allowed. However, this still allows passwords like boeing0, which I considered too easy to guess. We beefed up our password command by splicing in code from the password+ program developed by Matt Bishop. This disallows more passwords based on attributes of the user that might be found in the password file (such as phone number or mail stop) and also does a dictionary lookup. We have a large (200,000 entry) "dictionary" of 5, 6, and 7 character strings. We allow passwords that contain words 4 letters long or less. On a very infrequent basis (less than once a year), we run the crack program to see how many passwords we can guess. We enlarge the dictionary if it seems warranted.

One area of UNICOS security that probably has not received enough attention is denial of service attacks. This was amply demonstrated a month or so ago when one of our engineers froze up the system twice by filling the callout table with sleeps generated from ***rsh*** commands. It is unfortunate that nearly all of the UNICOS user limits are on a per job basis. With no limit on the number of jobs a user can spawn, these limits are not very restrictive. Per-user, system-wide limits on jobs and processes would go a long way to alleviating this problem.

So what about IRIX Security?

We have to admit that we have not gotten into security on our Origin2000 as much as we should have. At this time we certainly are not in a position to comment on its shortages or weaknesses. We can say what we have done so far and are planning to do.

We can say that, given the frequency of security alerts that SGI generates, they are very aware of and attentive to security concerns on IRIX and we feel that they are on top of things.

IRIX supports a security auditing mechanism much like the security log on UNICOS. We have set it up to record a small set of the events it can record. Any more than this seems to chew up an awful lot of disk space. We are logging little more than login attempts. We check these for failures from time to time. One shortcoming we have noticed is that records of failed logins do not contain the name of the originating host as they do on UNICOS. This is an invaluable piece of information when attempting to trace a breakin attempt. They also contain the name of invalid accounts that people have attempted to use. This is not a good idea. People too often get ahead of themselves and type in their password where the user name should go. Thus, what is recorded as an invalid user name, is in fact someone’s password. These should not be displayed. UNICOS merely puts out "INVALID" in these cases.

We currently have an ***rsu*** (see Section 3.2 on Human Process Error Prevention for a description of ***rsu***) on the Origin which requires no password. If you are a member of the proper group, executing ***rsu*** gives you root access immediately. This will be replaced by a real ***rsu*** or possibly the ***sudo*** program. The ***sudo*** command is a more standard Unix utility which allows certain users to perform certain privileged administrator tasks without actually becoming root.

There are currently no plans to port the UNICOS passwd enhancements to the Origin. The data separation scheme using the boeing group was installed on the Origin at the beginning, but there seems to be a problem on IRIX which will prevent the scheme from working. There is still a requirement that the parent directory of a user home directory be publicly accessible for a .rhosts file to be supported. We noticed in comparing the output from ls -l of /u on our IRIX with UNICOS that ba, bb, and bc have execute permission for others. This was done because of the .rhosts problem. If we have to leave /u opened up in this fashion, the scheme will not work.

In summary, UNICOS does provide a very high level of data security but Boeing needed additional items in order for it to meet The Boeing Computing Security Requirements. In all, the effort to enhance the UNICOS security was about six labor weeks.

We still have much to do to ensure we have adequate security in IRIX.

3.14 File Size Limits

There have been many instances when a process has created a very large file and exhausted the file system. Even DMF cannot cope with such occurrences. This situation can cause a great deal of work on the system to be lost.

The problem can, in large part, be resolved by enforcing a per file per process file size limit for those accounts that do not have any requirements for large files. This enforcement is applied equally to both interactive and NQS work and is controlled by the pfilelim[b] and pfilelim[i] values in the UDB.

It should be noted that files created by either ***rcp*** or NFS clients are not subject to the file size limit enforcement.

The NQS source code and librsc/setlimits.c were modified to support this feature.

This feature is desirable in IRIX as well.

3.15 CPU Time Limit Extension

Modified OS kernel to allow for more cpu time when process issued a SIGCPULIM. The vendors default value of 3 seconds was deemed far too small. Under UNICOS 10 this modification is no longer required as it can be defined with the Installation Menu System using XTRASEC.

IRIX should also provide the same capability.

3.16 User Controllable File Restore

This design describes a mechanism for allowing users to do their own file restores under the UNICOS system. Historically, when a user needed a file restored, he would call the data center help desk who would forward the request to a technology analyst. This made it difficult or impossible for restores to get done during evenings or weekends. Implementation of this design will speed up the restore process and free up valuable analyst time.

The functional requirements established for this feature were as follows:

- Users must have the capability of initiating a restore of any file they own at any time
- Users must be prevented from restoring files they do not own and from restoring files to any location other than where the file was when it was dumped.

- A catalog of all existing dump tapes must be maintained to facilitate locating the proper tape for use in the restore.
- Restores in progress when the system is taken out of service must be recovered or restarted from the beginning when production is resumed.
- Privileged users should be allowed to restore any file.
- User restores must be limited so as to prevent exhausting filesystems and monopolizing tape drives.
- A complete log of restore activities needs to be maintained so that problems can be resolved and a record of what tapes were actually used is available.

The feature was implemented in two pieces: Enhancements to the way base and incremental dumps are taken so as to produce catalogs of dumps, and a user interface to the ***restore*** utility, which will accept the user's input specifying which files are to be restored and then spin off one or more ***restore***s to do the work.

The ***dump*** enhancements make use of the **-A** parameter introduced at UNICOS 8.0. This flag is used to specify an alternate output file which can be read by ***restore*** to produce a listing of all the files written to ***dump***.

Our file system dump procedures were modified to run a ***restore*** in parallel with each ***dump***, so that a catalog file is written while ***dump*** is proceeding. Some minor modifications were made to ***restore*** so that the catalog will contain the file owner in addition to the file name.

A mechanism was added to the system startup scripts to recover any user restores in progress at the time of a scheduled or unscheduled system interruption.

It was necessary to implement the dump catalogs at least a month before automated restores were made available to the user so that a partial set of catalogs would be in place.

The user initiates a file(s) restore by executing the command ***urestore***:

urestore [**-f** filelist] [**-d** yymmdd] [**-b | -B**] [**-m** mail_address] [**-o**] [files]

The user can, if running interactively, specify no parameters, in which case the program prompts for a list of files or directories, and a date specifying the dump tapes to be used. The user can also specify a list of files on the command line or a file containing the list with the **-f** flag. File names can be absolute or relative: relative file names will be taken as relative to the current directory. The date can be specified with the **-d** flag. If run interactively, the program will request confirmation from the user after consulting the dump catalogs to make sure the requested files are on the specified tapes. If the specified files are not found on the requested dump tape, the user will be given a chance to specify another date. The user will be allowed to restore only those files and directories that he owns. If any specified files or directories already exist on disk, they will be removed before the restore is initiated. An interactive user will be given the option of skipping those files and directo-

ries. The user will be notified by mail when the restore completes. The **-m** flag can be used to specify an alternate mail address and **-m 0** can be specified to suppress mail notification. The **-b** option can be used to request that the most recent base dump tapes be used, while the **-B** option requests that the restore be done from the most recent base dump followed by all the subsequent inc dumps in sequence. This is to restore an entire directory that was deleted, but only some of the files in it were recently modified. The **-o** option requests that restored files overwrite any existing disk files with the same name. The interactive user will be asked whether this overwriting is to be done if the **-o** option is not specified. In a batch job, no overwriting will be done unless **-o** is specified

urestore will recognize a privileged mode based on membership in group tec, qa or help. A privileged user will be allowed to restore any file or directory.

An additional user command called ***urstat*** is supplied which displays the status of any restores the user currently has active. The exit status of this command is the number of restores that are still in progress. If the user supplies one or more request ids on the ***urstat*** command line, only those ***urestore*** requests are reported on.

The ***dump*** enhancements to produce dump catalogs were effected by modifying the local file system dump script. The ***dump*** statement in this script had the **-A** flag added to it, specifying that the alternate dump file is to be written to a named pipe. Just before the ***dump*** statement, the pipe is created with a ***mknod*** command, and a ***restore*** to read the pipe is launched into the background. After the ***dump*** completes, the pipe is removed. This is a little cumbersome, but the ability to pipe the ***dump*** output straight into ***restore*** with the "|" symbol requires enhancements to ***dump***. An additional flag, **-s**, is added to ***restore*** to request a table of contents that includes the owner of each file. The **-s** flag takes an argument, the filesystem mount point, which is needed to complete the path name of each file on the dump so that the owner can be determined. The catalog files are compressed with ***gzip*** and written to /wrk/dump_catalogs. The files are also marked with expiration dates so that obsolete ones can be removed on a regular basis.

The ***urestore*** command is simple line-oriented program that can be run from any terminal or workstation and does not require any graphics capability. When it is finished receiving input from the user, it spawns a script, running as root, into the background and terminates. This script initiates, in series, as many ***restore*** commands as are required to reload the requested files. It also writes a file into the directory /usr/spool/urestore for use during boot if the system goes down while the ***restore*** is in progress. The script removes the file when ***restore*** completes. As a final step, the script checks for migrated files that have been restored and does a ***dmput*** of each one to re-establish handles for soft-deleted files.

During the user input phase of ***urestore***, the program determines the set of files to be restored and makes sure the user owns them. The requested files are searched for in the dump catalogs that reside in the /wrk/dump_catalogs subdirectory corresponding to the appropriate filesystem(s). These catalogs, of course, are uncompressed using ***gunzip*** before being searched

The script launched by ***urestore*** has limits built into it to prevent too many ***restore***s taking

place at once. These limits are implemented through lock files in the /usr/spool/urestore directory. The limits are one per file system and three total. In addition, file systems with ***restore***s in progress are monitored for low-space conditions. Any ***restore***s in progress when a file system falls below the critical level as displayed by ***fsmon*** are suspended until the free-space reaches 5% above the critical level.

The startup script **/etc/rc.pst** is modified to launch a script which looks in /usr/spool/urestore for incomplete ***restore***s and relaunches them. This script first removes any files created by incomplete ***restore***s so that the ***restore*** will start off with a clean slate. Migrated files which were restored before the system interruption can cause a problem if they are left around when a second identical ***restore*** is started. This startup script waits 15 minutes after a boot before initiating any ***restore***s and also waits until the NQS queues are on and some tape drives are configured up.

The following provides possible problems and benefits with this feature.

The dump catalogs which are kept on /wrk (site specific file system) may be of substantial size even with compression. The disk space they consume should be closely monitored to be make sure it stays within acceptable limits.

There may be substantial time consumed by uncompressing the dump catalogs and scanning them for the requested files, especially if more than one catalog has to be scanned. Nevertheless, this is bound to be better than having no catalogs at all and having to guess what is on the tapes.

The method used for launching the actual ***restore***s (a separate script running as root) means that the ***restore***s will be done in a different UNICOS session than the one that executed the ***urestore*** command. This results in the user not getting charged for the resources used by the ***restore***s. This is unfortunate, but not a serious problem

The method used for creating dump catalogs requires that the file systems be mounted while ***dump***s are being taken. This may or may not present a problem for a site.

Once a restore operation has been spun off by the user interface, the user will no longer have control of it. There is no way he can cancel it. Intentionally killing a running ***restore*** and having to clean up partially restored files is not an appealing idea.

The existence of dump catalogs makes it possible to offer the user a facility for listing all of the files they have on dump tapes. The implementation of such a facility is not contained in this design but should be considered at a later date.

gzip and ***gunzip*** are GNU software and not part of the standard UNICOS system. There may be some difficulty in gaining the necessary permission to install them and make them part of the offering. However, tests have shown that they do a much better job, both in terms of cpu time and degree of compression, than either ***pack*** or ***compress***, and are therefore ideally suited for this feature.

This design assumes that the necessary dump tapes are always on site as apposed to say,

being located in a remote tape vault (for example, remote tape vaults used for disaster recovery purposes).

On the average, over the years, our users have used this feature 3 times per week.

We think this feature would be a useful addition to IRIX.

3.17 SDS and Ldcache Automatic Re-Configuration

The SSD solid-state storage device is a high-speed secondary memory. Data can be transferred between SSD and central memory at rates approaching 6500 Mbytes/sec using 4 VHisp channels. The SSD is an expensive piece of hardware and its investment needs protection by making its use efficient. Its capabilities provide i/o intensive codes with excellent performance allowing them to perform time critical work to meet demanding production schedules.

Programs use the SSD explicitly as SDS which is a particular way of configuring the SSD. The SSD can be configured as a disk device and/or as a secondary data storage (SDS) device. In addition the SDS can be further partitioned into cache (Ldcache) for disk resident file systems. While SSD configurations used for disk devices are static (meaning a reboot is required to change its configuration), those for SDS are not. The SDS can be configured dynamically and partitioned into user accessible SSD (i.e., SDS) and file system cache (i.e., Ldcache). Usage of SDS as Ldcache can benefit certain i/o intensive programs in a transparent manner.

Today our SSD is used for swapping (swap partition 0), user accessible SDS, and Ldcache. As one might imagine, the SDS is not always fully utilized whereas the Ldcache is in constant use. If, while the user accessible SDS is under utilized, more Ldcache were to be configured overall system i/o efficiency improvements may be obtained.

With the above in mind, we undertook a project to determine if there were indeed any benefits in dynamically reconfiguring the SSD device.

The approach taken involved gather and reviewing statistical historic data related to the amount of i/o traffic to the file systems followed by an evaluation report which reported the necessary changes to administer the SSD dynamic configuration and what the effect of this was in terms of increased machine throughput. The remainder of this Section presents the Evaluation report which includes some of the implementation and design and issues.

3.17.1 Evaluation report

The evaluation period was conducted over a period of four months (Aug - Nov 1996). The automatic configuration of Ldcache was performed only for our scratch file system; also known as the TMPDIR (/dev/dsk/work). User file systems were not included in the evaluation as a precautionary measure against file system damage that may have resulted from unscheduled system interruptions (i.e., our system was experiencing reliability problems).

The existing Ldcache configuration was shared among several of our file systems such as / root, /usr, /usr/spool, /local, and /work. We also at times would Ldcache the /usr/dmf (the DMF data base file system) to speedup the DMF process.

Prior to the evaluation the TMPDIR file system was configured with a static Ldcache size of approximately 40 Mwords. This configuration was changed to be dynamic in that the Ldcache size for TMPDIR was re-sized or re-examined every seven minutes with the intent of making the TMPDIR Ldcache have the lowest address of all file system Ldcaches. The lowest address was essential for allowing the TMPDIR Ldcache to be re-sized without having to relocated any other file system's Ldcache (an expensive operation). The TMPDIR Ldcache size was allowed to vary from a minimum of 40 Mwords and up to a maximum of approximately 1150 Mwords. The maximum value was set to always allow the largest user job SDS request of 700 Mwords to be immediately satisfied. Although this meant that as much as 700 Mwords of SDS could be unused at times when no user SDS requests existed, it precluded the necessity for developing sophisticated software to scan the NQS job input queue for SDS jobs and hold them while Ldcache for TMPDIR was re-sized. One other adjustment to the SDS auto configuration for TMPDIR, that had not been envisioned, was that it became essential to always provide a pad between what the users were actually using and the upper bound of the TMPDIR Ldcache. This allowed users' SDS arenas to expand immediately without being pre-empted, which is an expensive system operations in that it consumes large portions of the swap device and impacts the effective system swapping bandwidth, delays user SDS jobs unnecessarily, and can cause severe swap space fragmentation problems. The optimum pad size was determined to be 100 Mwords.

It was found that as user SDS jobs came and went and with others adjusting their SDS arena sizes, the SDS could become fragmented even though the system is designed to minimize this via the kernel SDS gravity-packing algorithm. To overcome this, a special program was developed that when executed would kick-start the kernel SDS gravity-packing algorithm to pack the users' SDS arenas prior to making adjustments to the TMPDIR Ldcache size. This technique has shown itself to work well.

Another unforeseen problem area was related to the releasing of operator held SDS jobs (ones held via the ***qmgr*** hold directives). If, when an SDS job (done automatically by the NQS dynamic scheduler described in a following Section) is released, the SDS physical space needed by the job was not available due say to the TMPDIR Ldcache size, then it would stay in "checkpoint" state and never be restarted. To avoid this situation a special interface was developed within the SDS auto configuration software to allow the TMPDIR Ldcache size to be minimized to 40 Mwords upon request and subsequently instructed to begin normal operations again at a later time. This technique has been shown to solve the problem.

The statistical data gathered from the historical ***sar*** data showed that most physical i/o to and from the TMPDIR file system were reads. The ratio between read and writes was approximately 2:1. A special TMPDIR Ldcache configuration was setup to determine to what extent the reads could be satisfied from Ldcache. It was found that much of the data

read from TMPDIR was read multiple times; this is ideal for a cache. With this in mind, the TMPDIR Ldcache was always configured to provide 4 times more Ldcache for reads than for writes (i.e., use of ***ldcache*** -h high, low was employed here).

Much effort was given to determine what Ldcache and physical i/o data to gather in order to find out the effectiveness of the new dynamic Ldcache for TMPDIR. The five minute ***sar*** sampled data was deemed to be so volatile that little could be learned. The re-read and re-write ratios ranged from 1.0 to thousands from one sample to the next. The most reasonable data collection period turned out to be monthly. This allowed the data to be presented in a more digestible form and provided an excellent indication of the SDS auto reconfiguration effectiveness in reducing the amount of physical i/o.

The monthly physical i/o for TMPDIR was plotted from February 1996 to March 1997. It showed that physical read i/o has been dramatically reduced starting with August 1996; the start of the evaluation period. This reduction held throughout the evaluation period and continues today.

One piece of data that would categorically indicate the usefulness of the SDS auto reconfiguration is the i/o wait time. Unfortunately this cannot be separated out for an individual file system such as TMPDIR. Thus we are left with the grand total for i/o wait time which is not useful as the total i/o is dominated by users' SDS i/o which typically is 100 times greater than any other form of i/o.

The assumption is made that if physical read and writes to TMPDIR can be satisfied from the Ldcache then i/o wait times are reduced. Our technology folks do not disputed this assertion.

An added bonus from this new Ldcache automatic configuration was realized when the file systems /tmp and /usr/tmp were eliminated and replaced with symbolic links to directories in the TMPDIR file system. These two file systems now enjoy the same Ldcache benefits as the TMPDIR file system.

The re-read and re-write Ldcache cache-to-disk ratios for TMPDIR prior to the evaluation averaged around 4.0 for reads and 1.0+ for writes. Today, these same two ratios are averaging 30.0 and 4.0 with the re-read cache-to-disk ratio hitting 200-300 at times. During the early part of 1996 the physical i/o to the TMPDIR disk was around 22 Terabytes per month. Today it averages around 7 Terabytes per month with the same kind of workload. This obviously has many benefits not only for system throughput but also on the physical stress imposed on the disks themselves

The logic for performing the SDS auto reconfiguration is done entirely by our Health Monitoring System (HMS) ***chkldcache*** script, which is described some in Section 3.24. This HMS script provides various informative log messages describing the actions it takes. These messages provide a feed-back loop for making improvements to the overall scheme.

3.17.2 Some further work or evaluations that could be conducted

1. Explore extending SDS Ldcache auto configuration to user file systems.
2. Explore ways to make use of all SDS space when no user SDS requests present.
3. Explore ways to adjust read/write SDS Ldcache partitioning using ldcache -h option.
4. Use a Benchmark Suite to evaluate various SDS Ldcache auto reconfigurations.

3.17.3 Concluding remarks

The Ldcache auto re-configuration has been shown to be effective in using what would be otherwise an expensive idle component of the system. With this facility, non-SDS users can now share the fast (and reusable) i/o characteristics of SDS with those employing SDS explicitly. The assertion is that overall i/o wait times are reduced with a corresponding improvement in system workload throughput. This assertion is largely support by ***sar*** data that showed i/o wait times have been dramatically reduced as compared to those prior to this enhancement.

3.18 Remote Shell (remsh) Enhancements

We find there is a large number of NQS jobs employing ***remsh*** during their termination phases. This is done for a number of reasons, the primary one being to simply notify the job owner of the job's completion status. Another is to actually provide a interaction between the NQS job and the owner executing on their workstation. While this operation may seem reasonable and simple, it does have serious adverse affects.

For example, if the remote shell hangs due to a remote system problem or if there is no one present at the remote system to respond interactively to some required action it can cause:

- Prolonged occupancy of an NQS execution slot (1-48 hrs had been observed)
- Job checkpoint failure when NQS system shutdown (Error 137 - open socket)
- Unnecessary job reruns due to checkpoint failures
- Overall work turn-around degraded
- Global NQS resources such as memory and SDS held longer than necessary
- Delayed job accounting which skews performance and capacity analysis
- Inefficient and non-productive use of data center time and labor

To deal with these problems the ***remsh*** was front-ended with a special script. The real ***remsh*** was renamed and the front-end script named ***remsh*** having ***rsh*** as a hard link. This design provided a transparent facility for intercepting the users' ***remsh*** so that an optional time-out could be inserted and for enabling a system shutdown signal to be caught. The time-out is used to deal with the case when ***remsh*** hangs unintentionally for too long on the remote system. By catching a system NQS shutdown signal the job can be successfully checkpointed and resumed by rescinding the ***remsh*** at shutdown and reinstating it during NQS recovery.

The default time-out is set to 1800 seconds and can be overridden by user with the environment variable REMSH_TIMEOUT. Additionally, the user can specify an executable file to be called when the shutdown signal is caught.

The enhancement to ***remsh*** solved the problems mentioned above.

This feature is view as being a useful addition to IRIX.

3.19 Memory Scheduling Enhancements

The UNICOS memory scheduling has a more dramatic influence on interactive work than its batch counterpart. NQS is used to schedule new batch work into the system for execution. Beyond this, NQS has no real control over this workload and the management, or performance of this work falls to the OS memory scheduling software. This OS software is tunable via the ***nschedv*** command.

After some years of struggling with the ***nschedv*** controls with limited success, and with the advent of our engineers requiring to run large memory interactive codes, we undertook the project of improving the OS memory scheduling to manage the evolving interactive and batch workload requirements.

This feature is covered in detail in another paper “The Age-Old Question of How to Balance Batch and Interactive” presented during the Stuttgart CUG 1998. The reader is directed to this paper for further information on this enhancement.

The requirement for *easily* balancing batch and interactive workloads is also needed for IRIX, especially on the Origin2000 system as it is far more likely it will have to deal with these workloads simultaneously while serving as a central shared computing resource.

3.20 Dynamic NQS Scheduler

The Dynamic NQS Scheduler provides a tool for controlling the initiation of NQS jobs based on priority. It allows control of multiple NQS queues and complexes, with a mix of different priorities in each queue or complex. This allows for better control of turnaround based on job priority.

The Dynamic NQS Scheduler is implemented as a single command, ***nqsched***, and a number of associated parameter files. ***nqsched*** operates by reading output from the ***qstat*** command, doing some computations based on its parameter files, and generating a number of ***qmgr*** directives. ***nqsched*** supports daemon mode, where both ***qstat*** and the ***qmgr*** commands are executed internally. It can also be run as a filter, accepting a ***qstat*** report as its input and generating ***qmgr*** directives as its output.

NQS input requests are categorized by request-priority into a small number (3) of “priorities”. These priorities are “CR”, “ME”, and “LO”. Jobs have an associated turnaround level of “A”, “B”, and “C”. Only jobs with levels “A” and “B” are considered to be “CR”

or “ME” priority by ***nqsched***. All level “C” work is considered to be “LO” for scheduling purposes.

nqsched supports control of queues, complexes, and groups. Queues and complexes correspond to their NQS counterparts, while a group is simply an arbitrary collection of queues or complexes. While ***nqsched*** cannot control a group directly, the ***nqsched*** “pro-rating” (see below) can be used to allow control of group members. The queues, complexes, and groups which ***nqsched*** controls are based on its parameter files. In particular, the ***nqsched*** definition of complexes should match the NQS complex definitions.

The ***nqsched***’s internal computations are based on a number of factors. For a given queue or complex, the computed limit is dependent on the following:

1. Factors from qstat report (dynamic)

    -Number of jobs in execution
    -Number of “CR”, “ME”, and “LO” priority jobs in input

2. Factors from parameter files (static)

    -Minimum and maximum limit for queue
    -Max limit if no “CR” jobs
    -Max limit if no “CR” or “ME” jobs
    -Weights for “CR”, “ME”, and “LO” jobs
    -General input job weighting factor

The limit for a given queue, complex, or group is computed in two stages. First, a weight is computed as follows:

```
wt =  #_of_running_jobs +
      #_of_LO_jobs * general_wt * LO_wt +
      #_of_ME_jobs * general_wt * ME_wt +
      #_of_CR_jobs * general_wt * CR_wt
```

Then the maximum and minimum limits are applied to the calculated weight and a limit for the queue/complex is determined.

The pro-rating option allows the computed limit for a complex or group to be distributed over the members of that complex or group. To prorate over a group, first the group’s limit is computed. Then the weights of each of the groups members is computed. Finally, the group’s limit is apportioned over the members of the group, based on each member’s computed weight. Prorating of a complex is done in a similar fashion. Prorating allows fine control over which queues receive execution slots, while providing global control over the total number of jobs running in the complex or group.

The ***nqsched*** daemon mode operates as follows:

Internally run ***qstat***

Calculate new limits
Internally run ***qmgr*** to set new limits
Sleep for an interval
<repeat>

Differing parameter files are needed over the course of a day. For example, LO jobs will be weighted more heavily during off-shift, and LO maximum limits will be relaxed as well. To provide an easy transition from one ***nqsched*** to the next, a simple rule is implemented - the most recently launched ***nqsched*** is the "real daemon", and any other will terminate automatically. This is achieved by means of an interlock file, which contains the process id of the most recently launched ***nqsched***. This scheme allows a new copy of ***nqsched*** to be launched by ***cron***, or by other means, without having to locate and explicitly signal the old copy of ***nqsched***.

3.21 System Panic Damage Control

Sudden system panics can cause a great deal of distress. This feature is designed to provide:

- Flushing of both system buffer cache (/bin/sync) and logical device cache (/etc/ldsync)
- A 'speedy' reboot of the system following the panic
- Notification to users on the system at the time of the system panic

The first bulleted item above is now provided by the vendor. The 'speedy' reboot is achieved mainly by having a recovery document that instructs a recovery analyst how to reboot the system following a system crash. This document is invaluable to the person who infrequently performs this activity.

Notification to users is paramount as data files could be lost or corrupted which will require the user to take special actions to recover their work correctly.

Immediately after the system reboot a program/script scans the lost+found directories in each file system and mails a notice to the owner informing them of lost files. Lost files are not moved to the user's home directory since this could possibly cause the file system containing the home directory to be exhausted. Instead, these lost files remain in their respective lost+found directories but under a hidden subdirectory. By moving them to a hidden subdirectory they will not interfere with subsequent system panics. The owner of the files are informed of the file location and that they will be retained for N days. After N days the files are removed.

Interactive users at the time of the system panic are informed that the system paniced. During the reboot while in single-user mode the /etc/utmp is saved as it is newly created by /etc/init when going to multi-user mode. When in multi-user mode the saved /etc/utmp is processed and all interactive users are mailed their notifications.

3.22 System Testing Environment

When the UNICOS system is taken out of production for new system testing or QAing, it is necessary to preserve the production state. Preserving production is achieved by checkpointing all NQS executing work and saving the checkpoint files and any queued jobs on disk. When the new system is brought up on the machine, a special NQS configuration is needed to avoid production work from being restarted. There are other known things, such as cron, accounting, and accessing/modifying files that were in use be checkpointed production jobs that should also be considered to avoid the testing from interfering with the production activities. This feature provides all necessary things to safe guard the production state.

This feature is covered in detail in another paper “The Good, the Bad, and the Ugly Aspects of Installing New OS Releases” presented at the Stuttgart CUG 1998.

3.23 File Expiration

This feature describes a scheme for implementing file expiration which was first introduced under UNICOS 7. It used an expiration timestamp stored in the inode site bits. Commands for displaying, altering or selecting files by time were extended to include expiration time. A script is run periodically to do the actual removal of expired files. This feature is primarily aimed at allowing users to easily designate when files have outlived their usefulness and can be automatically removed, thus reducing storage charges.

The functional requirements were as follows.

- Users must have the capability of specifying an expiration time for each file they own.
- All commands which can be used to display file times need to be modified to include expiration time.
- The **find(1)** command need to be modified to allow selection of files by expiration time.
- It must be easy to add or subtract file types and file systems from those affected by expiration.
- The script to remove expired files should be relatively simple and easy to modify to accommodate other selection schemes.
- The script which removes expired files must keep a log of all activities including any failures to remove a file and other errors.

File expirations are implemented by using the site bits in the file inode. Cray has reserved one word (64 bits) in the inode for sites to make use of in any manner they choose. Fifteen bits of this word is used to store the expiration date expressed as days after June 1, 1994. Where possible, it is treated like the other three times already maintained for each file: the last change time, last modification time and last access time. Unfortunately, the only mechanism Cray provided for manipulating the sites bits is the **fcntl(2)** system call. This call requires an open file descriptor. Thus, updating the expiration time on a migrated file would require recalling it in order to open it. Since this is considered completely unaccept-

able, a system call was needed to the kernel which allowed modifying the file expiration without opening the file. Cray provided a user exit for sites to add their own system call, beginning with UNICOS 8.0. Cray also planned to add a system call for generalized inode changes sometime later in 8.0. Cray did deliver this, so today this code is used instead and the kernel modifications introduced under UNICOS 7 is now discarded. Another slight oversight on Cray's part was the failure to have **restore(8)** maintain the sites bits when a file is restored. The system call for modifying a file's site bits is added to **restore(8)** to correct this defect.

The **fcntl(2)** mechanism provided for site bit manipulation is not be used at all.

A scheme similar to the one described in this design was implemented at Shell UK (now Shell Common Information Services). The developer, Andy Haxby, provided Boeing with the code as well as a copy of a paper he delivered at CUG describing the scheme and their experiences with it. A great deal of thanks are due to Andy for his help in getting this feature started and pointing out the likely pitfalls.

Since the file expiration is treated like the other three file times as much as possible, any command which allows the user to display or explicitly alter these times is modified to include expiration time. These commands are **ls(1)**, **find(1)**, **touch(1)**, and **fck(1)**. Until UNICOS 8.0, there were two different versions of **ls(1)**, both of which were be modified. The letter x was chosen to signify expiration for the modified commands. Thus, to list expiration times for all files in a directory, the command is

ls -lX

Files that have no expiration date have "Indefinite" displayed in the date column. Sorted output, i.e., the output from **ls** -lXt, lists files without expiration dates last. To find all files expiring more than *n* days from now, one would enter

find . -xtime +*n* -print

To find all files with expiration dates, one would enter

find . -expire -print

The only command the user will have for specifying file expiration is **touch(1)**. The syntax is

touch -x [*time*] *file*

where *time* is specified as mmddhhmm[yy]. If time is omitted, the expiration is set to indefinite. Alternatively, the user can specify

touch -X [*days*] *file*

where *days* is the number of days from now when the file is to expire. If days is omitted, it

defaults to 0, setting the expiration to indefinite. If the named f*ile* does not exist, it is created. Only the file owner or the superuser are able to change a file's expiration.

As mentioned, above, **fck(1)** is modified to display expiration. It displays expiration automatically if one exists, so no flag requesting is needed.

As previously mentioned, the key element in this design is a new system call which sets inode site bits. The kernel user exit introduced with UNICOS 8.0, **uesyscall(2)**, was backstitched into 7.0 to accomplish this. This system call provides for multiple uses by specifying a sub-system call number, a list of which is maintained in the include file uex.h. The call for modifying inode side bits will be known as **SETSBITS**. The system call will allow for modifying only a portion of the site bits, but no formal structure definition will be used in order to avoid changing any existing kernel code.

Both **touch(1)** and **restore(8)** will be modified to include this system call. The modifications to **ls(1)**, **find(1)**, **fck(1)**, and **nc1pfextr(1)** should be straightforward and will likewise parallel existing code wherever possible.

A script is run daily which removes expired files. Files are removed as soon as reasonably possible after expiration, usually within 24 hours. This runs the risk of no accurate backup of the file existing if it was modified shortly before removal, but it was decided that expiring the file when the user specified and precluding any additional storage charges were more important. The script of course keeps a log of all files removed including the owner, size, time of removal, and time of last modification. This should make it relatively simple to locate the correct dump tape should the user request that the file be restored.

A user has the capability to put an acl on a directory denying write permission to root. Should this happen, no extra measures are taken to circumvent the problem; the file will simply not be removed.

File expiration applies to all types of files including directories, which of course have to be empty to qualify. The expiration script only runs on user file systems. Attempts to place file expirations on other user accessible filesystems such as /tmp, /usr/tmp and /wrk are accepted but have no affect. Such files continue to be removed on a regular basis as they always have. An attempt to place an expiration date on a file on an NFS filesystem will result in an error.

The initial implementation was on a strictly voluntary basis. That is, only files that have an expiration date that the owner has added will be expired. Future changes could include an expiration scheme based on last modification or access time, wherein the expiration date would be used to prevent a file's removal after a period of no use. Changes such as this can be achieved by modifying the removal script only; no changes to system code will be required.

The problems associated with this feature are as follows.

The concept of making expiration time the same as the other three times associated with a

file breaks down at some point because not all files have expiration dates whereas the other three dates are attributes of all files. The solutions contained in this design for handling files without expiration dates are somewhat clumsy at best.

It would be unwise to expect very much disk space or very many DMF tapes to be freed up by this scheme. The Shell UK folks, using a more coercive scheme than what is proposed here, noticed an initial benefit, but that disappeared within a few months.

The real benefit comes from providing the users with a means for having files discarded automatically at a preset future date. For example, an engineer knows that support data for some design analysis needs to be kept ONLY for 2 years. This engineer may leave the company!

This feature needs to be part of IRIX.

3.24 System Health Monitoring and Automated Recovery Strategies

When Boeing introduced UNICOS 5.1 back in 1989 there were many unanswered questions about its reliability and robustness. Although we have seen a steady improvement in the UNICOS OS since then, the same cannot be said for some of its sub-components. Some of these sub components still pose problems for us today. During this time since 1989 we have developed a Health Monitoring System (HMS). The very first task for HMS was for managing the quirks of the NQS quickfile daemon; namely qfdaemon, which repeatedly would get itself into a cpu loop. This stole precious cpu cycles and stopped all SDS scheduling tasks causing very long job delays. From this one simple script, HMS has grown to over 36,000 lines spread among some 70 script files and a handful of small ‘helper’ programs. HMS maintains log files that not only indicate its findings and corrective actions, but also abnormal system behavior according to rule-sets. This data has proved to be immensely useful for post analysis of problems and simply for understanding normal system behavior so that the rule-sets can be improved.

We are asked frequently about the resources consumed by HMS. It turns out that on our T916/16512 system today HMS consumes, on the average, about 1 CPU-hour per day (that is only ~0.3% of the machine). All the benefits HMS provides us with are well worth this expense.

Some thoughts recently have been for integrating HMS concepts into the UNICOS Automatic Incident Reporting (AIR) facility. This may very well have been done, but with the recent T90P cancellation any further investment(s) need careful examination at this time.

HMS is not a static entity. It, by the very nature of new problems or situations thought to be impossible or unthinkable that arise almost every day, undergoes frequent enhancements. The ***sysedit*** feature described elsewhere in this paper helps with this administrative task.

Interestingly enough, the concept of an HMS has spread to other compute servers within the Boeing central data center based on the successes we have seen with it on UNICOS.

Whenever a new compute server service is being installed one always sees HMS as an early project.

HMS has provided us with a facility that greatly reduces the need for highly skilled people to continually monitor system activity for anomalies, and has in many cases provided automatic early detection of problems triggering recovery processes. Overall, HMS has been instrumental in helping with our efforts to improve the service quality and availability in a cost effective manner.

The primary goal and approach philosophy established for HMS were (and still are) as follows.

- Goal

  Provide an unattended and fully functioning system at all times

- Approaches

  Use shell scripts rather than program software to detect system failures
  Use predictive techniques to avoid system failures
  Emphasis placed on automatic corrective actions rather than simply reporting
  Log and report normal and perceived abnormal system behavior for analysis

There have been other CUG papers written on the automatic supervision of UNICOS in the past. All have addressed specific issues that help with promoting the concept of running the UNICOS system without requiring human intervention. The paper “Automatic Supervision of CRAY UNICOS Systems” by N. Attig, V. Sander, L. Wollschlager (KFA Germany) and R. Krotz (Cray Research GmbH, Germany) is a very good example. These papers, in general, concentrated on very narrow areas of system supervision such as system reboots or shutdowns, and automatic shutdowns triggered by unrecoverable hardware errors occurring on memory or disk. I suspect that these efforts have since been expanded to cover more system anomalies. The point being made here, is that there is evidence that the automatic supervision of UNICOS is considered a critical need by many. Obviously there is a need for such supervision which I suspect is driven as much by individual selfishness as by the more acceptable need to minimize service degrades.

Boeing’s HMS design philosophy is to execute ‘quietly’, correcting anomalous situations where ever possible, and only reverting to requiring human intervention when all corrective actions seem to be failing. The ideal would be for HMS to always know how to correct errors without ever having to resort to ‘outside’ help.

HMS is managed by a script file launched at system boot time. This script establishes the HMS environment and then enters a periodic loop during which it dispatches other scripts to perform specific ‘health checks’. HMS can be, at any time, be placed into a sleep state, woken up from a sleep state, terminated, or restarted. A logfile is maintained by HMS which records all findings and recovery actions. These log files are maintained on a daily

and monthly basis which rotate on a twelve-monthly basis. That is, there will always be one years worth of logfile data at any time. Typical logfile messages are shown below.

```
05/29/98 06:17:14 chkhardware: CPUs configured for i/o are 9 10 11
05/29/98 06:17:21 chksds: scanning /usr/adm/urm/Urm.980529 for urmd internal error
05/29/98 06:17:28 get_nqs_q_states: NQS BATCH queue Mlg_Ngt status_state changed to 'off'
05/29/98 06:17:29 get_nqs_q_states: NQS BATCH queue Mvl_Ngt status_state changed to 'off'
05/29/98 06:25:03 chknschedv: setting hog_max_mem <860160>blks for non-prime time
05/29/98 06:30:40 chkhardware: 1 CPUs missing from i/o cpu set - should be 4 i/o capable CPUs
05/29/98 06:30:43 chkldcache: SDS and Ldcache completely packed
05/29/98 06:30:45 chkldcache: no processes found using SDS
```

As the normal daily HMS logfile can contain many informative messages we found the need to augment the logging with another log which contained the important alert messages. The /usr/adm/syslog/HMSalerts_log was facilitated by making entries in the syslogd configuration file /etc/syslog.conf and using the /usr/bin/logger command to make the required HMS alert log message entries.

Our HMS software is contained on a PL and an hms.mk file is used to build and install the software ready for execution after modifications to the /etc/rc.pst and /etc/shutdown.pre have been made. HMS initiates from of /etc/rc.pst and shutdown from /etc/shutdown.pre.

HMS acts differently when run under 'production-mode' vs. 'QA test-mode'. This is done to avoid production files used by HMS from being disturbed by QA activities.

As mentioned above, HMS is large in terms of the number of shell scripts, and it is not possible to described all their actions in the context of this paper. However, the following list is provided to give the reader some idea of what HMS monitors, controls, and reports on.

- Starts all NQS queues during system startup ONLY after certain criteria has been met such as having tapes configured and available
- Reports on NQS job checkpoints and recovery status after system startup
- Captures NQS queue states so that same states preserved across system shutdowns
- Configures the tape subsystem; recovers tape subsystem if it fails
- Captures tape drive states so that same states preserved across system shutdowns
- Monitor and report on hardware configuration immediately after system startup such as number of active CPUs and CPU mode flag settings
- Reports if CPU downed automatically by system (i.e., CPU not exchanging in 30 secs)
- Monitor/validate cron entry updates - recovers from an erroneous update
- Verifies SSD configuration; validates SSD Vhisp settings; monitors URM behavior
- Ensures /etc/nschedv settings are correct; adjusts them per rule-sets
- Ensures sufficient user memory is available - warns if insufficient available

- Validate disk state and read/write modes; monitors disk errors; reports on jobs suspended due to unrecoverable disk errors
- Suspend NQS queues if disk space runs low; restart queues after problem resolved
- Makes predictions based on disk space usage rates for when disk space will be below a critical threshold - very useful in preventing catastrophes
- Verifies that correct file systems are mounted
- Verifies file ownership and permissions for top-level file system trees; if invalid NQS will be stopped until problem corrected
- Monitors the user command directories to ensure nothing is missing
- Monitor for correct behavior of system demons - stops/restart them as necessary
- Tracks and modifies Ldcache based on SDS usage
- Monitor network resources such as mbufs and mbuf denials
- System date/time checks
- Compute and reports disk/filesystem working set space - typical report shown below.

File System	Workspace Size (GB)	# reg files in Workspace	Workspace files %	# reg files in filesystem	Workspace Ratio
u	20.278854	34790	2.830%	1229527	0.840
u1	6.427071	1715	0.420%	408040	0.533
u2	8.491650	5551	1.030%	537052	0.704
u3	21.227905	19188	4.510%	425835	1.760
big	88.715290	519	0.710%	73001	1.439
i	4.412514	4126	2.000%	206172	0.732

Top 90% usage of 7-day workspace for /u

Workspace Size (GB)	# reg files in Workspace	Workspace files (%)	# regular files owned	User Names
7.001152	684	2.06%	33238	abc1234
1.555645	1352	2.14%	63302	xyz4321
1.548332	22193	46.69%	47533	dce9876
0.988781	1588	9.40%	16887	opq4567

- Report/Warn on file system inode usage rates - typical report shown below.

A-O-K --->201 days remain before depleting i-nodes for /u
  Action required: None
A-O-K --->562 days remain before depleting i-nodes for /u1
  Action required: None
A-O-K --->527 days remain before depleting i-nodes for /u2
  Action required: None
A-O-K --->33 days remain before depleting i-nodes for /u3
  Action required: None

```
A-O-K --->1591 days remain before depleting i-nodes for /big
  Action required: None
A-O-K --->204 days remain before depleting i-nodes for /i
  Action required: None
```

It is true to say that HMS on UNICOS has reached the state in its evolution that today our support analysts look to HMS for providing strong indications the system is operating correctly. The HMS log is typically viewed for signs of trouble immediately after system boot. This surely is a good sign that HMS has indeed been worthwhile.

Should SGI/Cray provide HMS-type software ? Probably not, as every customer site will have its own ideas as to what constitutes ‘system health’. However, a similar feature to the UNICOS AIR facility would make an excellent foundation in IRIX for sites wishing to develop their own HMS facility.

4.0 Summary

The basic UNICOS system has evolved over the years and now offers a rich set of features to allow a site, such as Boeing, to come close to successfully administering a centrally located shared computing facility such as the T916. We have found that it is still necessary to refine these as well as adding our own ones in order to satisfy our unique requirements. As with UNICOS maturing over the years, so have our local enhancements also matured, and we now have a system that requires very little attention on a day-to-day basis. That is not say there is nothing more to do. Our system continues to mature from one problem to the next with the MTTP getting larger and larger!

With UNICOS 10 being the last UNICOS major release it will be difficult to convince SGI/Cray to add more features or to refine existing ones based on customer inputs.

One thing has become crystal clear to me over the years. Experience with dealing with vendors is that they are loathed (maybe this is too strong a word) to spend too much time refining their software to meet every last request from their customers. Understandably, it is expensive for them to do so. This leaves an individual site, wanting additions to the system, in a difficult position. Much of this difficulty can be reduced if the system source code is available to them. With this, they can immediately make their own changes while also pursuing the vendor to make like changes (providing the vendor with the code changes can help with this endeavor). Adding features in this fashion however, needs to be done with great care as the long term maintenance of the local code can become very burdensome. Source code also provides another benefit. When problems arise or if a site needs to make enhancements, the source code provides an experienced person with the necessary information to understand and resolve problems or to arrive at a good design for the enhancements. Without the source code these things become almost impossible to carry out, and the site is at the mercy of the vendor.

The user exits provided in TMF with UNICOS 10 is a win-win situation. SGI/Cray should explore this idea to its fullest extent in IRIX.

My hope is that other member sites will strongly encourage SGI/Cray to give importance to their own requirements and to have them incorporated into the IRIX system as it is developed to manage our future products such as SN2 and SV2.

5.0 Acknowledgments

Much of the background material given in this paper came from my associates at Boeing within the Technical Services organization. I wish to extend my thanks to them all in providing their insights and for their aging and dust covered design documents. In particular I want to thank Mark Lutz and Bill Matson for putting up with all my questions that no doubt stretched their memories, and patience, at times.