Basic Administration [5]

This chapter describes the following tools and methods, many of which are commonly used in the day-to-day operation of a UNICOS system:

- Using the `cron` and `at` utilities
- The temporary directory (`TMPDIR`)
- Communicating with users
- Monitoring system security
- Job and process recovery
- Kernel user exit (`uesyscall`)

Warning: This chapter contains warnings and information critical to the proper use of a Cray ML-Safe configuration of a UNICOS system.

5.1 Using the `cron` and `at` utilities

The `cron`(8) and `at`(1) utilities are invaluable tools for automating many administrative tasks. You can use them to run administrative tasks at regular intervals and during off-peak hours, when they will not interfere with the interactive work of most users. Neither utility is appropriate for every administrative task; by using both, however, administrators can avoid many time-consuming and repetitive tasks.

Note: For information on using the `cron` and `at` utilities on a Cray ML-Safe system configuration, see Section 8.4.4, page 201.

5.1.1 Administrative use of `cron`

The `cron`(8) process executes commands at specified dates and times. As a system administrator, you can use the `cron` process to run tasks on a periodic basis. The `cron` utility is especially convenient for the following administrative tasks:

- Turning off the programs in certain directories during prime time (by using the `chmod`(1) utility to remove execute permission for the program).

- Running programs during hours other than prime time for the following procedures:
  - File system administration
  - Accounting
  - System security procedures

You can specify the commands to be executed by using the `crontab`(1) utility, which takes as its argument a `crontab` file that describes the commands and the times at which they should be run. The `cron` process consults the files located in the directory `/usr/spool/cron/crontabs` to determine which tasks are to be performed and at what times they are to be performed.

Each user maintains only one individual `crontab` file, which is the user's ID, and which requires that the `/usr/spool/cron/crontabs` directory contain separate files for each user. The name of the `crontab` file is used as a user ID to get user and group permissions.

A `crontab` file consists of lines containing six fields each. The fields are separated by spaces or tabs. The first five fields are integers that specify the following:

- Minute (`0` to `59`)
- Hour (`0` to `23`)
- Day of the month (`1` to `31`)
- Month of the year (`1` to `12`)
- Day of the week (`0` to `6`, with `0` = Sunday)

Any of the first five fields can be an asterisk (*), indicating that any value is appropriate. The last field is the command line to be executed at the appropriate time. For example, the following line in a `crontab` file would execute a local program called `/usr/bin/task` once a week, every Sunday morning at 6:30 A.M.:

```
30 6 * * 0 /usr/bin/task
```

If the system is not running at the time a command is to be executed by `cron`, the command does not execute. Consequently, `cron` is most appropriate for periodic tasks that do not interfere with normal system operation if not executed, and for tasks that must be run at a specific, regular time.

The cron(8) process is usually started from the file /etc/rc during system startup. See crontab(1) and ksh(1) for more detailed information.

The cron daemon makes an attempt to report fatal errors that cause termination by printing an error message on the system console. Also, a user of the at or crontab utility receives a warning message if the cron daemon is not active at the time the at or crontab command is issued.

The cron daemon can limit dynamically the number of concurrently running jobs. It can also maintain up to 26 separate queues, and control the number of jobs executed in each queue. The file /usr/lib/cron/queuedefs is used to maintain definitions for all queues. If this file does not exist, the default values are used. See queuedefs(5) for more detailed information.

Changes to queue definitions take effect before the next job is executed by the cron daemon.

The cron daemon logs all command invocations, terminations, and status information in the file /usr/lib/cron/log. Records that begin with the character > pertain to command invocations. Two invocation records are written for each command execution: the first displays the command being executed; the second contains the login name of the user who executed the command and the process ID, job queue, and time stamp for the command. Command termination records begin with the character < and are similar to the second invocation record, except that a nonzero termination status or exit status is also printed. Records that begin with the character ! indicate status information.

The cron utility uses named pipes to communicate between the user-level commands and the daemon process.

5.1.2 Administrative use of at

The at(1) utility submits commands or shell scripts for execution at a specific time. Its format is as follows:

at *time* [*date*]

It takes as its argument the time (and optionally, the date) at which to execute a list of commands that it reads from the standard input. Other options and arguments are available (see the at(1) man page).

The at utility is intended primarily for single, nonrepetitive execution of a command or script, and is thus especially appropriate for scheduling large jobs to run during off-peak hours. However, the at utility can set up periodic execution of a task if a script being run by at uses at itself to reschedule its

own execution. For example, if a script, whose file name is `/usr/bin/task`, contains a line such as the following, it reschedules itself to be run at 3:30 the next morning (and the morning after that, and so on, because the script still contains the rescheduling line):

```
echo "sh /usr/bin/task" | at 0330 tomorrow
```

The advantage to using `at` instead of `crontab`(1) for periodic command execution is that you are assured that the command will run; if the system is down at the time `at` would normally run the command, it runs as soon as the system is brought up again. The drawback to this automatic execution is that the command is not guaranteed to run at the specific time you request. That is, if a command is submitted through `at` to be executed at 3:30, and the system is down for dedicated time until 7:30, the command will run at 7:30 when the system is running, which may interfere with users' work if it is a large, CPU-intensive command.

The prototype file allows you to customize `at` command files by controlling what information is written into the `at` job file, `/usr/spool/cron/atjobs`. If a file named `/usr/lib/cron/.proto.`*q* exists (*q* is a queue name), this file is copied into the job file. Otherwise, the `/usr/lib/cron/.proto` file is used.

The following substitutions are made during creation of an `at` job file:

Variable	Description
`$a`	User's current account name
`$m`	User's current file creation mask (see `umask`(2))
`$l`	User's current file size limit (see `ulimit`(2))
`$d`	Name of the current directory
`$t`	Time (in seconds since 1/1/70) when the job is scheduled to execute
`$<`	Read standard input until EOF is reached

The following is an example of a prototype file:

```
newacct $a
cd $d
ulimit $l
umask $m
$<
```

At minimum, a prototype file containing `$<` must exist to successfully run the `at` utility. The `at` utility exits with an error if no prototype file exists.

The `at` utility can queue jobs in one of 25 different queues, with the `cron` daemon controlling the number of executions for each queue. (You can use this queuing mechanism to limit the use of the `crontab` and `at` utilities.) Running the `at` utility with the `-qx` option as the first argument queues the command in queue `x`. The default queue is `a`. A special queue, `b`, is defined as a batch queue; jobs in this queue run whenever the defined maximum level is not exceeded (as specified in the `queuedefs` file). Queues `d` through `z`, by default, run at the same priority as `b`. Queue `c` (available with the standard AT&T `at` utility to run `cron` executions) is not available with the UNICOS `at` utility; the `crontab`(1) utility should be used to submit `crontab` jobs. Jobs in all other queues run at the time specified on the command line.

5.1.3 Restricting use of `crontab` and `at` utilities

Users can potentially abuse system resources when using the `crontab`(1) and `at`(1) utilities. However, both the `crontab` and `at` utilities provide methods for restricting user access. The `/usr/lib/cron/cron.allow` and `/usr/lib/cron/at.allow` files contain the login names of users (one per line) allowed access to the `crontab` and `at` utilities, while the `/usr/lib/cron/cron.deny` and `/usr/lib/cron/at.deny` files contain login names of users denied access to the utilities.

When a user submits a `crontab` file, `crontab` checks `cron.allow` for a list of users permitted to have a `crontab` file. If no `cron.allow` file exists, the file `cron.deny` is scanned for users who are denied `crontab` files. If neither file exists, only `root` is allowed to have a `crontab` file. The same process is used for determining access to the `at` utility. The null `cron.allow` file would mean no user is allowed a `crontab` file, while a null `cron.deny` file would mean that no user is denied a `crontab` file.

For additional information on the `at.allow`, `at.deny`, `cron.deny`, and `cron.allow` files, see the *UNICOS Configuration Administrator's Guide*, Cray Research publication SG–2303.

5.2 The temporary directory (`TMPDIR`)

The `TMPDIR` directory contains temporary user subdirectories and files. `TMPDIR` is created at the beginning of an interactive session or batch job. All the files and directories in `TMPDIR` are deleted at the completion of the session or job.

UNICOS commands and libraries create temporary files in `TMPDIR` instead of `/tmp` or `/usr/tmp`.

UNICOS temporary directories are owned by the user and have group and other permissions turned off. This prevents other users from seeing or deleting files in a temporary directory they do not own.

5.3 Communicating with users

During the operation of a UNICOS system, it is frequently necessary for administrators to use the system to communicate information to its users. This section discusses a number of UNICOS commands and tools that enable you to communicate with users:

- The `wall`(8) command
- The `/etc/motd` file
- The `/etc/issue` file
- The `/usr/news` directory
- The `write`(1) utility
- The `mail`(1) utility

5.3.1 The `wall` command

The `wall`(8) command broadcasts items of immediate concern to all users currently logged in to the system. Run the command by typing the following:

```
/etc/wall
```

The `wall` command responds by telling you to type your message and to press `CONTROL-d` when you are finished. To ensure that all users who are currently logged in see a message sent by `wall`, run the command while you have `root` privileges; otherwise, the message goes only to users who allow messages to be written to their terminals (see `mesg`(1)). Additionally, users who are not currently logged in will never see the message; `wall` is thus not a suitable method for communicating a message to all users who have accounts on the system.

The `wall` command is typically used to send the following messages:

- Warnings that the system will soon be brought down for scheduled downtime. Users who log in after the message is sent, however, miss the message and should be notified by the `/etc/issue` file (see `login`(1)).
- Warnings that the system must be brought down immediately to address a system emergency.
- Warnings that a particular file system has run out of disk space and that users should make an immediate effort to delete any unneeded files (see the description of the `-g` option on the `wall`(8) man page).

5.3.2 The `/etc/motd` file

The `/etc/motd` (message-of-the-day) file is displayed to users after they are logged in to the system. The `/etc/motd` file is an ordinary text file, and the administrator may place messages in it by using any UNICOS text editor.

Messages that should be placed in `/etc/motd` are those that are less immediate than those requiring the use of `wall`(8), but they are important enough that users should be forced to see them. The administrator should remove messages from `/etc/motd` as soon as they are no longer needed. Suitable items for inclusion in this file include the following:

- Warnings to users to clean up unnecessary files on a particular file system or systems
- Brief explanations of recent problems that may have affected a number of users, often with a pointer to a news item containing a more detailed explanation

5.3.3 The `/etc/issue` file

The `/etc/issue` file is displayed while a user is logging in, before the user has successfully logged in to the system. It is an ordinary text file, and you may place messages in it by using any UNICOS text editor.

Messages placed in `/etc/issue` should be brief and so important that users may need the information to decide whether or not to log in to the system. Possible messages include the following:

- Warnings that the system will be brought down soon (so that users who do not see a `wall`(8) message are not surprised when the system is brought down shortly after they log in)

- Warnings that the system is being used for dedicated time and that not all users will be able to log in

5.3.4 The /usr/news directory

When users log in to the system they are alerted to the existence of any new files placed in the /usr/news directory. When a user then runs the news(1) utility, it displays any news files that have been created or modified since the last time the user ran news. The files placed in /usr/news are ordinary text files created with any UNICOS text editor, and they are usually assigned names that give a general idea as to their contents. For instance, a news file containing information about a modification to a system library might be given the name new.library.

Because users are not notified of the existence of a new news file until the next time they log in, and because there is no guarantee that any given user will see the file (a user may choose to ignore the item by not running the news utility), /usr/news is appropriate for items that are not time-sensitive or items that are of interest to only some of the system's users. These categories include the following:

- Notices regarding recent system changes, such as a newly installed version of a command or library
- Explanations of imminent system reconfigurations or changes
- Explanations of recent system problems and their possible effects on users

It is a good idea to remove any old files in /usr/news periodically, not only to save disk space, but also to prevent new users on the system from having to read through a long list of out-of-date news items. The /usr/news file may be cleaned out regularly by cron(8).

5.3.5 The write utility

The write(1) utility initiates immediate person-to-person communication with a logged-in user by opening that user's tty or pty for writing and copying each line of text you type to his or her screen. To write to a user with a login name of dolores, for example, you would issue the following command:

```
write dolores
```

If the user `dolores` happened to be logged in on more than one tty or pty, you could specify the connection:

```
write dolores ttyp001
```

If, in this example, the user `dolores` is currently logged in, a message appears on her screen indicating that you are writing to her. Typically, the user `dolores` replies by writing back to your account; each line of text she types appears on your screen.

Given the immediate nature of its communication, the `write` utility allows you to perform the following functions:

- Converse with a user
- Obtain information about what a user is doing
- Warn a specific user to stop what he or she is doing
- Instruct a specific user to clean up his or her directories

Because each typed line appears on the other user's terminal without regard for what that person may be typing at the moment, it is easy for the other user's messages to your terminal to appear to interfere with your typing. This problem is customarily solved by having the two users take turns typing, ending a message with an `o` on a line by itself (standing for "over," much as in a two-way radio conversation). To end such a session, either user then ends a message with an `oo` on a line by itself (for "over and out"). Thus, a typical "conversation" carried out by `write` might look like this (your input appears in **bold**):

```
# write dolores
Message from dolores (ttyp001) - Mon May 11 08:20:15 - ...
Yes
o
Please clean up your account, we're out of space.
o
All right, I will.
o
Thank you.
oo
<EOT>
```

Because many users either do not know of this etiquette when using `write`, or do not follow it, they think that `write` is difficult to use. In practice, it is used

rather sparingly, mainly when more convenient forms of communication (such as simply calling the user on the telephone) are impossible. Taking steps to educate your user community in the proper use of the `write` utility will prove valuable when `write` is the appropriate communication method.

Note: On a UNICOS system or Cray ML-Safe configuration, for `write` to execute properly, the user's active security labels must be equal.

5.3.6 The `mail` utility

The `mail`(1) utility provides a way to leave messages for specific users, whether or not they are currently logged in to the system. The `mail` utility is used as follows:

```
mail ralph
```

Type in message

```
CONTROL-d
```

You may specify more than one account name, in which case copies of the message go to each user named. The next time users to whom you (or anyone else) have sent mail messages log in to the system, the system alerts them to the fact that they have mail messages waiting. The `mail` utility is thus particularly well suited for messages such as the following:

- Instructions to clean up directories
- Asking or responding to questions
- General communication

In theory, there is no guarantee that the recipient of a mail message will actually see the message, because the recipient may choose not to run the `mail` utility to read the message; however, in practice, most users read their mail when they log in.

Note: On a Cray ML-Safe configuration, the recipient of a mail message might not be authorized to read mail at the classification with which it was sent.

For more information see `mail`(1) and `mailx`(1).

5.4 Monitoring system security

Maintaining security on UNICOS systems is largely a matter of vigilance on the part of the system administrator, who should maintain constant surveillance for potential security problems and for evidence of past security breaches. Fortunately, the UNICOS system includes programs that provide the necessary tools for the creation of a set of procedures that allows you to automate much of the daily work of monitoring system security. This section discusses security issues in three areas: system security (ensuring that the super-user privileges are safe), user security, and partition security.

5.4.1 Super-user privileges

In the UNICOS operating system, with `PRIV_SU` enabled, the user identification number (user ID) of `0`, associated with the account named `root`, has special privileges and may override the security features governing the activity of normal users. Such a user is referred to as a *super user*, and the super user's powers allow the administrator great flexibility in responding to system problems and keeping the system running smoothly. The dominant security concern for a UNICOS administrator is ensuring that access to super-user privileges remains solely in the hands of the administrator and the administrator's staff. Failure to guard this access allows an unauthorized user to acquire super-user privileges. At best, one user could then look at other users' sensitive files without authorization and, at worst, an outside intruder (knowingly or unknowingly) could cause damage to the entire system.

5.4.1.1 Password security for super user

The password to the super user (`root`) account is the first line of defense against security breaches. Anyone logging in as `root` or using the `su`(1) utility to acquire super-user privileges uses this password.

Cray Research recommends the following steps to maintain secure access to the `root` account:

- The `root` password should not be obvious and should be very difficult to guess. Do not use a normal word in any language that might be known to a majority of the system's users. Additionally, capitalizing a random letter or two (not the first letter of the password), or including a punctuation character or a numeral in the password, or both, helps to keep super-user privileges safe from an intruder who is trying to guess the `root` password.
- The `root` password should be changed frequently, at least once a month.

- The `root` password should never be written down anywhere.
- The `root` password should be known to as few people as possible. Generally, these should be the system administrator and the administrator's staff.

Use of the `root` password can be monitored, and potential security breaches caught, by compiling the `su` utility so that it logs each use of the utility in the `/usr/adm/sulog` file. The administrator can then use the `grep`(1) utility to generate periodic lists of successful and unsuccessful attempts to assume super-user privileges by use of `su`. These lists can be compared against the names of users known to have valid authorization, alerting the administrator to unauthorized super users (a security breach) or users who are repeatedly trying to gain super-user privileges (a security risk).

5.4.1.2 Physical security

A person with access to the SWS, OWS, and IOS consoles and a knowledge of how to halt and reboot the system could do so and thus acquire unauthorized super-user privileges.

To guard against this possibility, Cray Research recommends that the SWS, OWS, and IOS consoles and the system itself be physically accessible only to those persons with genuine need for that access. If this is not possible, they should at least be monitored to prevent unauthorized persons from attempting to enter commands on the system console.

5.4.1.3 setuid programs

An executable UNICOS program may have the setuid bit in its permissions code set, indicating that whenever any user executes the program, the program runs with an effective user ID of the owner of the file. Thus, any program that is owned by `root` (user ID `0`) and has its setuid bit set is able to override normal permissions, regardless of who executes the program.

This feature is useful and necessary for many UNICOS utilities and commands, but it can be a potential security problem if an astute user discovers a way to create a copy of the shell owned by `root`, with the setuid bit on. To avoid this possible security breach, the administrator should make regular checks of all disk partitions on the system for programs that have a setuid (or setgid) of 0.

The `find`(1) utility can generate a list of all setuid/setgid 0 files on the system (if all file systems are mounted), as follows:

```
find / \ -user 0 -perm -4000 -o -group 0 -perm -2000 \ -print
```

This list may be compared against a list of known setuid/setgid 0 programs. Any new setuid/setgid 0 programs that are not on the known list and whose creation you cannot account for may indicate a security breach.

The administrator should check the list of known setuid/setgid 0 programs regularly to ensure that none have been modified since the last check and that any modifications that have been made are known (in other words, were made by the system administrator or a member of the administrator's staff). Unknown modification of a setuid/setgid 0 program may indicate a security breach.

Finally, the list of known setuid/setgid 0 programs should be checked to ensure that write permission on each file is properly restricted.

Because checking the entire system for setuid/setgid programs uses a large amount of CPU time, Cray Research recommends that this check be performed during off-peak hours. Use of the `cron`(8) or `at`(1) utility to perform the check automatically and to notify the administrator of any suspicious results should make the task unobtrusive.

5.4.1.4 `root` PATH

The `PATH` environment variable consists of a list of the directories searched by the shell for typed commands. This means that the `PATH` for the `root` account must have the following security features:

- It must never contain the current directory (.).
- All directories listed in the `root PATH` must never be writable by anyone other than `root`.

The `root PATH` is set in two separate places:

- The `/.profile` file sets the `PATH` for `root` whenever `root` logs in on the system console.
- The `su`(1) utility changes the `PATH` after a user has successfully entered the `root` password to assume super-user privileges.

Both places should be monitored from time to time to make sure they have not been changed since the last approved change known to the administrator.

Keeping the current directory out of the `root PATH` is somewhat inconvenient; super users must remember to precede the names of any programs or scripts they want to run from their current directory with `./`, as in `./newprogram`, because the shell does not search the current directory for a command name.

However, convenience should not take precedence over system security. Failure to follow these guidelines leaves the system open to a security breach.

For example, suppose a knowledgeable user creates a program that mimics a commonly used system utility, such as `ls`(1). In addition to performing the expected system function (listing the files in the current directory), the new `ls`(1) utility makes a copy of a program such as `ksh`(1) and turns on the setuid bit on the copy. An unsuspecting super user with the current directory in `PATH`, having changed directories to a user's directory and inadvertently run the bogus `ls`, then creates a setuid 0 shell, which gives anyone executing it complete control over the system.

5.4.2 User security

In addition to general system security, the administrator should ensure that files owned by system users are secure from examination and modification by other users.

5.4.2.1 The `umask` utility

The system default umask value is normally set in the `/etc/profile` file by using the `umask`(1) utility. It allows you to choose the permissions that will typically be set when users create new files. For example, a umask value of 027 means that the group and other write permissions and the other read and execute permissions are not set when a user creates a file. For possible umask values and descriptions, see the `umask`(1) man page.

In general, only the owner of the file should have write permission, which makes a default umask value of 022 appropriate. If members of a given user group should not be able to read the files of other user groups, using a umask value of 026 to remove other read permission is recommended.

You should choose a umask value that restricts default access permissions to a level appropriate to the desired security of the system. However, because users can override the default value by using the `umask` utility themselves, do not make the default umask value too stringent, as users may find that the default value interferes with their work. For instance, if two users are working on a joint project, and each needs access to the other's files, they may want to change their umask values so that, on any new files they create, the permissions will be more open.

5.4.2.2 Default PATH variable

The default PATH variable for the system's users is set in the /etc/profile and /etc/cshrc files. It specifies the system directories that will be searched for command names typed by the users.

The users expect to be able to execute programs in the current directory without having to precede the program name with ./ to explicitly indicate the current directory. However, many UNICOS systems traditionally place the current directory first in the PATH, which can make the users vulnerable to a security breach, as described in Section 5.4.2.4, page 121. The current directory should thus be the last entry in the default PATH, after the normal system directories.

5.4.2.3 User groups

User security can be enhanced by the careful placement of users into groups. In general, it is a good idea to use factors external to the system when deciding upon the placement of users into groups. Some examples might be the following:

- Members of a specific software project
- Accounts for a client company purchasing system time
- Intercompany divisions

Having many groups, each containing a small number of users, is safer than having fewer groups, each with large numbers of users with access to each other's files. Members of most logical groups (for example, members of a software development project) want to share files with one another, and the default umask should permit this.

To prevent inappropriate sharing of data, you should create a group with only one user in it, rather than create a default "other" or "miscellaneous" group for users who do not fit elsewhere. Because users may belong to more than one group, and groups are active simultaneously, you may also choose to create a separate group for each individual user at the time you create the account, and then add users to additional logical groups as necessary.

5.4.2.4 File-owner fraud

Neither the listed owner ID of a file nor its location in the directory tree always leads to the actual creator and owner of the file. That is, users tend to think of the files residing in their home directory as their only files, even though they may own files in another home directory, such as those being used for a project

involving several other users. Conversely, it may not be completely appropriate to count files that reside in one user's home directory tree but are owned by another user.

Users may realize this confusion and try to avoid a disk usage monitoring system by using the `chown`(1) utility to change the ownership of some of their files to another user (most likely one who will cooperate and give the file back when requested). Nevertheless, `diskusg`(8) and `du`(1), when used together, provide a general idea of the users who are perennial problems.

5.4.2.5 Login attempts

Unauthorized users might attempt to gain access to the system by making repeated attempts to login. To help prevent such attempts, you can configure the number of bad login attempts that will be allowed before the `login` terminates. By default, the system will allow an unlimited number of bad login attempts. To put a limit on such attempts, edit the `/etc/config/confval` file (see `login`(1)).

> **Note:** For information on limiting login attempts on a UNICOS system or a Cray ML-Safe system configuration, see Chapter 8, page 145.

5.4.3 Partition security

When administered properly, the UNICOS file system should provide adequate protection for user and system files. You can enhance system security, however, by mounting partitions only when they are needed. In particular, if there are users who will be allowed dedicated time on your system, you can provide extra protection for those accounts by not mounting the file systems that contain other users' accounts.

To prevent users from accessing disk partitions directly, without going through the UNICOS file system, the disk device nodes in `/dev/dsk` and `/dev/rdsk` must never be readable or writable by anyone other than `root`.

5.5 Job and process recovery

This section describes the recoverability considerations and restrictions of the UNICOS operating system.

5.5.1 Restrictions to job and process recovery

This section lists restrictions to recovering jobs and processes submitted either interactively or as batch jobs via the Network Queuing Environment (NQE) and Network Queuing System (NQS). The sections that follow list restrictions common to batch and interactive recovery, and restrictions unique to batch recovery.

5.5.1.1 Restrictions common to batch and interactive

The following list describes restrictions common to batch and interactive job and process recovery:

- All the files that a process was using when it was checkpointed must be present when the process is restarted. This includes all open files, the present working directory, and any shared-text binaries (such as shells) in use by the process. If any of these is not available when the process is restarted, the `restart`(2) system call fails and returns an `EFILERM` error (errno 51). In the restart file, each of these files is identified by the file system minor number and inode number. If either number changes, for example, if a file system is restored after a process is checkpointed, the `restart`(2) system call fails with an `EFILERM` error.

  The requirement for shared-text binaries to be present can cause restart failures if a system is booted on an alternate root file system, for example, when a system is upgraded from one UNICOS update release to another. If the old root file system is not available when checkpointed jobs are restarted, the restarts will fail with an `EFILERM` error, because the shells are not available. When converting from one root file system to another, the old root should be mounted on some alternate mount point (`/mnt` for example) so that checkpointed processes can be recovered.

- If the `RESTART_FORCE` option is not specified on the `restart`(1) invocation, any file that was in use at checkpoint time must not have been modified since that restart file was created in a nonsequential fashion. That is, the `restart` will fail if any bytes in the file between offset 0 and the file size have been modified since the checkpoint occurred. This rule allows the job or process to be checkpointed and to continue execution, sequentially extending output files, without invalidating the restart file.

- The access permissions of the files and directories in use at checkpoint time must not have been changed from their original values, or changed to deny the required access at restart time.

- User and group ownership of the files must be unchanged.

- The sum of the sizes of all unlinked files in use by the target process set must be less than the system limit specified as `MAX_UNLINKED_BYTES` in `config.h`. Note that the intent is to checkpoint and restart processes and jobs using unlinked (sometimes called *zerolink*) files, as long as the files in question are small. This covers such cases as unlinked files created by `/bin/sh` in order to handle here documents, and small temporary files typically used by compilers. Applications that use very large temporary files (for example, `assign -t`) should be changed to use files in the linked temporary directory.
- If a user attempts to copy a restart file or to move a restart file to a different file system, it is no longer marked as a restart file and is not restartable.
- An open pipe connection that originates, or terminates, outside the checkpoint target set prevents the successful completion of the checkpoint.
- Any form of TCP/IP socket usage prevents the successful completion of the checkpoint. Note that the innocent use of certain TCP/IP system calls, such as `gethostname`(2), can cause the TCP special device `/dev/net` to be opened, and this causes a checkpoint request to fail.
- At checkpoint time, there must be sufficient disk space to contain the restart file. Note that the restart file contains at least two sectors worth of header information, any pipe data in transit between processes, the contents of any unlinked files in use, and the user control structures and program image for each process in the checkpoint target set.

  On systems with SSDs, any secondary data segment (SDS) regions in use are appended to the data region of the associated process.
- There must be sufficient system resources, such as inode buffers, file table entries, and job table slots.
- There must be sufficient process slots available so that the successful execution of `restart` will not exceed either the system or user maximum number of processes allowable.
- None of the process, process group, and job IDs needed for the successful execution of `restart` may be in use at restart time.
- Processes using shared memory segments (CRAY T90 series systems only) cannot be successfully restarted.
- For a UNICOS system or Cray ML-Safe system configuration, the user must be the owner and have MAC write access or be an authorized user or privileged process.

5.5.1.2 Recovery restrictions unique to batch

The following list describes job and process recovery restrictions unique to batch jobs submitted through NQS:

- If the user specifies at submission time, by way of the `qsub`(1) option `-nc`, that the job is not to be checkpointed, it is never checkpointed, even in response to a `qchkpnt`(1) command.
- The super user cannot checkpoint an NQS job directly (using the `chkpnt`(1) utility), without letting NQS know what is happening, and expect it to recover. The `qmgr`(8) command must be used to checkpoint an NQS job so that the main NQS daemon is aware of what is happening.
- Because NQS does not use the `RESTART_FORCE` flag on the `restart` invocation, any file that was in use at checkpoint time must not have been modified in a nonsequential fashion since the restart file was created. That is, the `restart` will fail if any bytes in the file between offset 0 and the file size have been modified since the checkpoint occurred. This rule allows the job to be checkpointed and to continue execution, sequentially extending output files, without invalidating the restart file.

5.5.2 Checkpoint and restart errors

If something goes wrong during a `chkpnt`(2) or `restart`(2) system call, an error code is returned in the global variable `errno`. For lists of such error codes, see the `chkpnt`(2) and `restart`(2) man pages.

The following section discusses how to use the `crash` program to examine the restart-information buffer to help determine why a checkpoint or recovery was not successful.

5.5.2.1 Examining the restart-information buffer

If you are having difficulty determining why an application will not checkpoint or restart, attempt to recreate the scenario on a relatively quiet system, and use the kernel debugger program `crash`(8) to examine the restart-information buffer.

The system is built with several restart-information buffers. A restart-information buffer is obtained and used for each checkpoint and restart operation.

The `resinfo` subcommand of the `crash` command may be used to obtain a usage summary for the restart-information buffers in a system. Using a quiet

system allows you to examine an individual restart-information buffer without having it overwritten. If you have determined which restart-information buffer you want to see in detail, invoke `crash` and issue the following command to receive a detailed account of everything that is still in the restart-information buffer. The `-` (dash) option causes the long form of the listing to be output.

`resinfo -` *buffer number*

If you are not sure which buffer to examine, the `resinfo -` command displays all the restart-information buffers.

5.5.3 Recovery and signals

The UNICOS operating system supports the automatic checkpoint and restart of batch jobs run by NQS across multiple shutdown and restart events. No modifications are needed for batch job requests run by NQS in order to take advantage of the UNICOS recovery facility, except in special circumstances.

There are two signals involved in the implementation of job and process recovery. The `SIGSHUTDN` signal warns of impending system shutdown, and the `SIGRECOVERY` signal is delivered to a recovered process. The following sections discuss these two signals.

5.5.3.1 `SIGSHUTDN`

To support special batch job requests that cannot be automatically checkpointed and restarted, and to allow limited recovery of interactive processes, a user process can register to catch the `SIGSHUTDN` signal. This indicates that the system is in the process of an orderly system shutdown.

Upon receipt of a `SIGSHUTDN` signal, the catching process can take steps to record its own state for later recovery, or improve its chances for recovery by closing unrecoverable files, and so on. Interactive processes can checkpoint and kill themselves by using the `chkpnt`(2) system call, thereby creating a restart file for later recovery.

The NQS `qmgr`(8) command for bringing down NQS in an orderly manner is as follows:

`shutdown` *grace-time*

The *grace-time* operand specifies the length of time between notifying a job with the `SIGSHUTDN` signal, and checkpointing or killing the job. Note that a

`shutdown` or `shutdown 0` command does not send the `SIGSHUTDN` signal to the jobs; rather the jobs are immediately checkpointed or killed.

Any shutdown scripts that are executed when an orderly system shutdown is imminent should first invoke the NQS `qmgr` command to shut down NQS, checkpointing all recoverable batch jobs, and then send the `SIGSHUTDN` signal to all interactive processes in the system by using the `killall`(8) command.

5.5.3.2 `SIGRECOVERY`

When a process is recovered from a restart file, either by itself or as a member of an NQS job, a `SIGRECOVERY` signal is posted to that process. By default, the `SIGRECOVERY` signal is ignored, so a process must register a signal handler for `SIGRECOVERY` if there may be action necessary (for example, to restore previously closed unrecoverable files).

5.6 Kernel user exit (`uesyscall`)

The `uesyscall` system call is a user exit into the kernel that allows you to write a site-specific system call. This function gives you access to kernel structures not otherwise available and allows a site to implement functionality in the UNICOS operating system that requires kernel support.

Warning: The kernel user exit (`uesyscall`) does not meet the requirements of a Cray ML-Safe configuration of the UNICOS operating system.

The structure of `uesyscall` is as follows:

```
int uesyscall (int subsyscall, void *paramaddress, int len);
```

subsyscall — Sub-system call number defined by the site in `uex.h`, the system call include file. The sub-system call allows multiple system calls from one common entry point.

paramaddress — Address of the parameter list passed to the system call. The site-defined parameter list allows different parameters to be passed to each sub-system call.

len	Length (in words) of the parameter list. The length argument allows different parameters to be passed to each sub-system call.

An entry in the system call table has been added to the `uts/c1/os/sysent.c` file to support the `uesyscall` system call. The `uts/c1/md/krn_uex_syscall.c` file contains the system call source. The associated `uts/include/sys/uex.h` include file contains user exit definitions. The source for the system call and include file are distributed with source and binary releases.

The `krn_uex_syscall.c` source file contains a `stub` routine that simply returns. The main routine parses the input parameters and calls specified sub-system calls, which allows you to write multiple site system calls.

If you want the system call to be accessible to any user, it is recommended that you write a library interface to the system call. Creating a library interface allows extra sanity checks and validation, and provides a cleaner, more understandable user interface.

Caution: Use caution when creating a site-specific system call to avoid introducing the ability to corrupt data and to panic the system. In addition, the UNICOS kernel is now multithreaded. If a site adds code to update any tables in the kernel, you may need to place multithreading locks around the kernel structure being updated.