The *Try* System
– or –
How to Avoid Testing Student Programs

Version 1.11

Kenneth A. Reek
Undergraduate Computer Science Department
Rochester Institute of Technology
Rochester, NY 14623-0887

ABSTRACT

The **TRY** System, which runs under UNIX[1], allows students to test their programs in a controlled manner with the instructor's test data. The system will accept any type of files from the student, and the instructor has great flexibility in determining what is to be tested and how the tests are to be performed. The results of the tests are recorded in a log file for later grading. At no time does the student have direct access to the test data or the log file. The instructor is protected from damage that might be caused by malicious student programs.

March 13, 2000

---

[1] UNIX is a registered trademark of The Open Group.

Table of Contents

1. Overview

The **TRY** system allows students to test their projects in a controlled manner with the instructor's test data. The instructor creates a shell script to build the student's project and a script to run the project. When the student runs **TRY**, the student's file(s) are copied into a scratch directory, and the instructor's scripts are executed to create the project and test it. The results of each attempt are recorded in a log file for that student. Additional scripts can be created to give the student information about a test that the project failed, or to give the student information from the log file.

There are no limitations on the type or number of files that **TRY** will accept—it simply copies all of the files named by the student into the scratch directory. The instructor determines how many and what kind of files are appropriate for a project by the way the "build" script is written. Similarly, the only preconceived notion that **TRY** has of how a project should be tested is that there may be more than one test, and the tests are numbered from 1 to 127. The instructor determines how many and what kind of tests should be performed by the way the "run" script is written.

Each time the student runs the **TRY** program, an entry is added to that student's log file indicating the results of the attempt. Possible results include: unable to build the project, failed a test, and so forth. The instructor may give information to be added to the log file to further explain an error. The log file is formatted with automated processing in mind.

The **TRY** program runs as a set-user-id program so that it can access the instructor's test data and scripts and the log files even though the student cannot directly access them. Execution of student programs in the instructor's account, a gaping hole in the security of other automated grading systems, is not a problem with **TRY**.

2. How TRY Works

TRY may be used to test any type of program in any language, shell scripts, makefiles, etc. The word "project" will be used in this document to refer to the student's work.

When a student thinks his project is finished, he invokes **TRY** as in the following example:

```
$ try kar-grd lab3 makefile *.c *.h
```

The first two arguments are given to the student by the instructor, and specify the account in which the project is to be tested (the instructor's grader account is most appropriate) and the title of the project (called "project-code" in this document). The remaining arguments are the files that make up the project.

TRY looks in the **.tryrc** file in the specified account for an entry whose first field is the specified project-code. The remaining fields specify the "try directory" and the "project directory" to be used for the project. **TRY** moves to the try directory, creates a scratch directory for the student if one is not already present, moves to the scratch directory, and copies all of the files named on the command line. (The scratch directory is normally cleaned out after every run.)

If there is a file **news** in the try directory or in the project directory, the news is given to the student. A record is kept in the student's home directory to prevent old news from being reported every time the student uses **TRY**.

From within the scratch directory, **TRY** runs a shell script provided by the instructor to build the student's project; this may do compilations, make, formatting, or whatever is appropriate. If the project was successfully built, **TRY** runs a shell script to run the student's project. This script is also provided by the instructor, who is free to test the project in any way he or she sees fit.

If a test fails, another shell script is executed to give the student information about the failed test, presumably to assist him in fixing his program. The instructor may allow or disallow this, or may limit or expand upon the information the student is given, as this script is also written by the instructor.

A separate log file is maintained for each student, and the result of each run of **TRY** is recorded here. If the student gives the **–l** (the letter *el*, not a one) option to **TRY**, a shell script is run which prints information from his log file. The instructor writes this script, and can choose to give as much or as little information as is appropriate.

With two exceptions, all of the shell scripts provided by the instructor are executed with the instructor's user-id. This makes it easy to write scripts that access files in the project directories. The **build_safe** and **run_safe** scripts, however, are executed with a nonexistent user-id. These scripts are used when commands given by the student must be executed (e.g. when using a student-supplied makefile). The nonexistent user-id protects the instructor from mischief that might be caused by running these commands with the instructor's user-id.

2.1. .tryrc

The **.tryrc** file contains basic information about projects. It resides in the home directory of the account in which student projects will be tested. An instructor's grader account is the recommended choice, although an instructor's personal account is also acceptable.

The file contains an arbitrary number of entries, one per line. Blank lines and lines that start with a pound sign (#) are ignored, and may be used to put comments in the file. Each entry has the following format:

project-code try-dir project-dir options

The fields are separated by white space, and the third and fourth fields are optional. Long entries can be entered on multiple lines by ending each line except the last with a backslash. Leading white space on the continued lines is skipped.

The first field specifies the *project-code* that the students will use in the **TRY** command. The second field indicates the directory where **TRY** is based for this project (see *The Try Directory*, below). The third (if present) specifies the project directory (within the try directory) that contains the scripts and files needed for testing the project (see *The Project Directory*, below). This is a relative pathname that is interpreted from the try directory; the pathname may include subdirectories. If the third field is absent, the project directory name is assumed to be the same as the *project-code*.

The fourth field, if any, specifies options that apply to this project. In order to give options for a project, a project directory must be given. If several options are desired, they are given as a comma-separated list. The list may not contain unquoted white space.

2.1.1. .tryrc Options

The currently-implemented options are listed below.

off Submissions are no longer accepted for this project. This is useful to prevent late submissions while still allowing students to read news, do queries, or examine their log files for a project.

dummy
This project-code is a dummy entry which does not correspond to any project. Queries, news and log file dumps are allowed. This might be used as a way to allow a student to dump his entire log file.

save Do not clean out the scratch directory when **TRY** finishes. The files are left in the scratch directory, possibly for future examination by the instructor (e.g. when debugging your shell scripts). Should **TRY** be run by the student again, the scratch directory will be cleaned out at the beginning of that run and the saved files will be lost. The **save** option is therefore reliable only when used on the last project to be done for some time in the try directory; in this event, the **cleanup** script should be used to delete any unneeded files to conserve disk space.

ignore
Signals will be ignored during runs with this project. See also *Signals*, below.

post Signals will be posted during runs with this project. See also *Signals*, below.

initstatus
The exit status of the **init** script is checked; if non-zero, **TRY** logs the failure and exits.

cleanupstatus
The exit status of the **cleanup** script is checked; if non-zero, **TRY** logs the failure and exits.

yyyy Changes the date format in log file entries to yyyy/mm/dd (the default is yy/mm/dd).

NAME=value
Sets the environment variable *NAME* to the given value. If the value contains white space or commas, the entire value (not just a portion of it) must be quoted with either ´ or ". Environment variables can be helpful in getting one set of shell scripts to work for several different projects, with the environment variables triggering different actions for different projects. Environment variables given values in this way take precedence over variables set by **TRY** (see *The Environment* below for a list of these).

The following four options control the locations of various things in the try directory; it may be useful to return to this section after reading *The Try Directory*, below.

log_dir=*dirname*
This specifies a directory in which the log files should be put. If a relative pathname is given, it is interpreted relative to the try directory. The default value is ".". This value is given in the environment variable **LOG_DIR**.

script_dir=*dirname*
: This specifies a directory in which common scripts will reside. The default value is ".". A relative pathname is interpreted relative to the try directory. This value is given in the environment variable **SCRIPT_DIR**. (See note 3 of Appendix 1.)

wd_dir=*dirname*
: This specifies a directory in which the scratch directories should be put. A relative pathname for this is also interpreted from the try directory. The default value is ".". This value is given in the environment variable **WD_DIR**.

wd_backpath=*dirname*
: This specifies the pathname needed to get back to the try directory from within a scratch directory. The default value is "..". This value is given in the environment variable **WD_BACKPATH**. This option is required whenever the **wd_dir** option is used. As an example, with `wd_dir=Scratch`, the appropriate backpath would be `wd_backpath=../..`

TRY deletes all environment variables beginning with "TRY_". This allows the instructor to use variables with this prefix in scripts without worrying about whether the student's environment contains them already.

2.1.2. Default Options

If the **.tryrc** file contains an entry whose project name is identical to the try directory given for a group of projects, the options given for this dummy entry will apply to all projects in that try directory. Note that the dummy entry must have all of its fields given; any values may be given as they are ignored. It is suggested that the name / be given for the try directory and project directory to prevent students from accidentally using this entry as a real project. Note that the **dummy** option must not be given, or it will apply to all other projects in the same try directory.

If an option is preceded by **no_**, that option will be turned off for a project. This gives the instructor a way to turn on default options that are usually used, and turn them off for specific projects for which they are unwanted. (This mechanism cannot be used to turn off the setting of environment variables.) Figure 1 shows a **.tryrc** file illustrating the use of options.

2.2. The Try Directory

All of the information accessed or maintained by **TRY** for a set of projects is kept in a directory called the "try directory." This includes:

a) the scratch directories (one for each student) in which student projects are actually built and tested,

b) the log files (one for each student) in which the result of each try is recorded,

c) the default optional scripts **dump_log** and **explain**, and

d) the project directories, which contain the files needed to test a particular project.

The locations of the scratch directories, the log files, and the optional scripts may be modified by using the appropriate options in the **.tryrc** file (see *.tryrc Options*, above).

The instructor may choose to put each project in a separate try directory; in this case, every

```
try/309 /       /              log_dir=Logs,wd_dir=Wds,wd_backpath=../..
all     try/309 .              dummy
1-1     try/309 week1/hello    off
1-2     try/309 week1/sep.comp
1-3     try/309 week1/uc.lc    no_log_dir
2-1     try/309 week2/copy_n   OPT=2
2-2     try/309 week2/storage

try/325 /       /              save,ignore,script_dir=Scripts
325-1   try/325 proj1          off
325-2   try/325 proj2          no_save,post
325-3   try/325 proj3          log_dir=/home/fac/kar/pub/not_a_good_place
```

Figure 1
.tryrc file using options

student would have a separate log file for each project. Or, many projects can share one try directory, and each student's log file would then contain entries for all of the projects.

It is recommended that the try directory be created in a directory whose mode disallows public (and possibly group) access. The try directory itself must have at least mode 111 (see *The "Safe" Scripts*, below); the restrictive mode of the parent directory would then prevent students from accessing any files in the try directory.

2.2.1. Project Directories

All of the files and shell scripts needed to test a project are placed in one directory called the "project directory." This must minimally include a shell script to build the project and a shell script to run the project. Data files, output files, other scripts and any other files needed for this project also live here.

2.2.2. Scratch Directories

When a student runs **TRY**, a scratch directory called "wd.*account*" is made for that student, where *account* is the account name the student is using. By default, the scratch directories are made in the try directory; if a **wd_dir** is given in the **.tryrc** file, then the scratch directories are created in that directory. Each student has a different scratch directory. In this way, different students can be running **TRY** at the same time with no conflict.

Because running more than one **TRY** at a time in the same scratch directory can lead to unpredictable results, **TRY** attempts to determine whether the scratch directory is in use. This is done with a lock file called ".tryid" in the scratch directory. If a conflict is found, the student is told to interrupt all the **TRY**'s and start over, though they can continue executing if they so choose.

In order to guarantee a known environment in which to build and run the student's project, the scratch directory is cleaned out before the student's files are copied. It is also cleaned out after **TRY** has finished unless the **save** option is given for the project. Note that subdirectories and files whose names begin with a dot are also deleted. Symbolic links are deleted, but their targets are not.

TRY creates a file in the scratch directory called ".account"; this file contains the account number of the student using the directory and may be accessed from the instructor's shell scripts. This may also be obtained from the **STUDENT** environment variable.

2.2.3. The Log File

A log file called "log.*account*" is created for each student, where *account* is the account name that the student is using. By default, the log files are made in the try directory; if a **log_dir** is given in the **.tryrc** file, then the log files are created in that directory. Log files are created with mode 400, but **TRY** preserves their modes if they have been changed. An entry is appended to this file for each run of **TRY** indicating the account number used, the date and time that **TRY** was invoked (as opposed to when it finished), the *project-code*, and the status of the attempt.

Log file entries have the following format:

account project date time status

Fields are separated from one another by a tab except that the date and time fields are separated by spaces. The time is stored in hh:mm:ss format. By default, the date is stored in yy/mm/dd format, and there are four spaces between the date and the time. If the **yyyy** option is given, the date is stored in yyyy/mm/dd format and there are two spaces between it and the time.

Figure 2 lists the values that may appear in the status field. *Message* is text that is passed back by the instructor from the build or run script with the **try_log** program. *Signal name* is the name of the signal that caused **TRY** to abort. If **TRY** fails due to missing or incorrect scripts, or to an internal error, an appropriate message is written to the log file (these messages are not listed here).

The first entry in each log file contains

`Log file created for` *name*

in the status field, where *name* is the student's name from the password file. The project field in this entry will be "!", and the time will be earlier than any other entry in order to ensure that this entry will remain at the top after a sort.

2.3. Copying Student Files

TRY accepts only file names, not directory names. To submit all of the files in a directory, shell meta-characters would be used.

When a pathname is given for a file, the copy of the file is made using only the last component of the pathname. This ensures that, regardless of the pathname given, the copy resides in the scratch directory. If, however, a student gives two pathnames whose last components are identical, then both files will be written to the same copy, and one

```
Log file created for student's name
Completed.
Completed: message
Failed build.
Failed build: message
Failed build_safe.
Failed build_safe: message
Failed test n.
Failed test n: message
build script could not be executed.
build_safe script could not be executed.
run script could not be executed.
run_safe script could not be executed.
Interrupted: signal name
```

Figure 2
Log file status values

will be lost. For example,

```
$ try kar-grd 4-2 p2.c misc/p2.c
```

both files would be copied to "p2.c" in the scratch directory, and only the second one would survive. If this situation occurs, the student is informed and **TRY** is aborted and no log entry is made.

When **TRY** copies the student's files into the scratch directory, they will be owned by a nonexistent user id. The public and group permissions of each file will be set the same as the owner permission, however, so that they can be accessed from the instructor's scripts.

2.4. Instructor-Supplied Shell Scripts

TRY uses shell scripts provided by the instructor to build the student's project and to run it. These scripts reside in the project directory. Common scripts may be shared among many projects by creating links from one project directory to another. Some optional scripts may reside in the try directory or the optional script directory, and be shared in that way. Appendix 1 contains a table that summarizes the purposes and requirements of all the various shell scripts.

There are two general categories of scripts, protected and unprotected. Unprotected shell scripts are executed with the instructor's user-id, and should be used only when all commands to be executed were given by the instructor. Protected shell scripts (the "safe" scripts) are executed with a nonexistent user-id, and are used when it is necessary to execute commands given by the student, e.g. executing a student's shell script, or using *make*(1) with a student supplied makefile.

Because of the way in which **TRY** executes them, Bourne shell scripts must begin with the line

```
#! /bin/sh
```

and C shell scripts must begin with

```
#! /bin/csh
```

or

```
#! /bin/csh -f
```

This does not apply to System V Unix.

Due to a bug in the C shell, it cannot be used for the **build_safe** or **run_safe** scripts unless all of the directories above the scratch directory have public read and execute access. For reasons of security, the parent of the try directory usually has no permission for anyone other than the owner. The choice is between giving up this security to use the C shell, or retaining the security by using the Bourne shell.

Executable programs may be used in place of any of the shell scripts described. While they are less convenient to write, programs run more efficiently than shell scripts.

All of the scripts described are invoked by **TRY** from within the scratch directory. In order to access files in the project directory from one of these scripts, the pathname from the scratch directory to the project directory must be used, as in the following example:

```
$WD_BACKPATH/project-dir/name
```

where *project-dir* is the name of the project directory. **TRY** defines an environment variable

called **PROJ_DIR** whose value is the name of the current project directory. The example below shows how this variable would be used in one of the scripts:

```
ln $WD_BACKPATH/$PROJ_DIR/x-o x.o
```

Use of this variable makes it easier to share common scripts among several projects.

2.5. The Unprotected Scripts

These scripts are appropriate when the only commands to be executed are those given by the instructor.

2.5.1. The init Script

This script is optional. If it exists, it is run prior to the **build** or **build_safe** script to perform any initialization that must be done before the student's project is built. The user's file names are passed as arguments.

This script is most frequently used to perform tasks that could not be done in the beginning of the **build_safe** script (because of its being executed with a nonexistent user-id). It can also be used to perform initialization unique to a particular project so that a common **build** script could be used, or to perform authentication of the student's account.

The exit status of the **init** script is ignored unless the **initstatus** option is given. In this case, a non-zero status causes **TRY** to log the failure and exit, and the **try_log** program may be used in the **init** script to add a more detailed explanation to the log file.

The following example shows an **init** script that copies files from the project directory into the scratch directory. The file modes are changed to make them accessible to the nonexistent user-id with which the **build_safe** script is run. This avoids the need to allow public read access to files in the project directory. An environment variable is created that indicates the number of tests that were found (see *Modifying the Environment*, below for more details).

```
#! /bin/sh

for N in 1 2 3 4 5 6 7 8
do
   cp $WD_BACKPATH/$PROJ_DIR/*.$N .
   chmod 666 *.$N
done
echo N_TESTS=8 > .ENVIRONMENT
```

2.5.2. The build Script

The purpose of the build script is to create the student's project from the file(s) that were copied. This may involve compilation, formatting, running *make*(1), and so forth. It may also involve doing nothing, for example when the student is supposed to submit an executable file for testing.

The instructor may call his build script either **build** or **build_safe.** **TRY** places no restrictions on what can be done in these scripts, however it is risky to execute certain commands in the **build** script. See *The Safe Scripts* below for a detailed discussion of the differences between them.

When the build script is invoked, the names of the files that were copied are passed as arguments. *echo*(1) commands may be put in the script to inform the student of what is happening in the script or to explain errors. The build script should return an exit status of zero if the project was successfully built, and nonzero if something went wrong. **TRY** tests this exit status; if a nonzero status is returned, the failure is recorded in the log file and **TRY** stops.

If the build script ended successfully, **TRY** creates a file called ",newest_file" that contains the name of the most recently modified file in the scratch directory. This is useful when the name of the file being built is not known ahead of time, as might be the case when executing *make* in the **build_safe** script.

The umask is set to 077 while the **build** script runs in order to prevent temporary files created by compilers and other programs from being accessed by other users. After the script finishes, **TRY** changes the modes on all subdirectories and files to make the public and group permissions identical to the owner permissions. This ensures that the files can be accessed from either the **run** or **run_safe** script.

The following build script compiles a single Pascal file; if all goes well, the file **a.out** is created. Because the *pc* command is the last command in the script, its exit status is returned by the script.

```
#! /bin/sh
pc $1
```

The next script expects an arbitrary list of C source and header files, and compiles them with a main program and print function supplied by the instructor. It also echoes the commands so that the student will know where any errors that are printed originated. Note that the instructor's object files are named *file***–o** rather than *file***.o**; this is to avoid

the automatic deletion of old object files that occurs on some UNIX systems. The *rm* command ensures that no object files sent by the student survive.

```
#! /bin/sh
rm -f *.o
D=$WD_BACKPATH/$PROJ_DIR
ln $D/main-o main.o
ln $D/print-o print.o
echo cc *.c *.o -o lab2
cc *.c *.o -o lab2
```

The build script in Figure 3 compiles a student function that is supposed to duplicate the work done by the library routine *strncpy*. The script uses *nm*(1) to check the student's object file to make sure that *strncpy* is not called.

Figure 4 shows a script that expects two file names, one C source file and one header file. It does a lot of checking that the names are correct, and then compiles the source file.

2.5.3. The run Script

The purpose of the run script is to test the student's project. This might involve running a program several times with various input files and comparing the output that was produced with correct output, running *make*(1) to test the student's makefile, running the student's shell script, running a program to check some aspect of the student's file(s), and so forth.

The instructor may call his run script either **run** or **run_safe**. **TRY** places no restrictions on what can be done in these scripts, however it is risky to execute certain commands in the **run** script. See *The Safe Scripts* below for a detailed discussion of the differences between them.

The only preconceived notion that **TRY** has about testing projects is that there may be many separate tests, numbered in the range 1–127. If the student's project passes all of the tests, the run script should return an exit value of zero. If a test fails, the script should return the test number as an exit value. Depending upon this value, **TRY** will record the success, or the test number that failed, in the log file.

Exit status values 129–255 may also be used for failed test numbers 1–127. **TRY** ignores the high order bit, but passes it to the **explain** script, where it can be interpreted in any desired way. An **explain** script illustrating this is shown in the next section. The **run** script should not return an exit status of 128.

It is important that **try_run** be used to execute the student's program. **Try_run** limits the cpu time, file space and other resources that can be used by the program (on System V, only the file space can be controlled), but it also prevents the program from going outside of the scratch directory. If the student's program were executed directly in the **run** script, it would be able to access, create and delete files in the instructor's account just as the instructor could.

The umask is 0 while the **run** script executes so that any files it creates will be readable by subsequent scripts. If it is necessary to create files outside of the scratch directory (e.g. by running a compiler or other utility that creates temporary files),

```
#! /bin/sh
cc -c $1
if [ $? -ne 0 ]
then
   exit 1
fi
names=`nm -g *.o | egrep ´_strncpy$´`
if [ -n "$names" ]
then
   try_log Used library strncpy
   echo It is cheating to use strncpy from the C library!
   exit 1
fi
ln $WD_BACKPATH/$PROJ_DIR/main-o main.o
cc *.o
```

Figure 3
Build script that checks for library functions

```
#! /bin/sh
if [ $# –ne 2 ]
then
   echo You gave me $# files, but I was expecting 2. Please try again with
   echo one .c file and one .h file.
   exit 1
fi
cfile=  hfile=
for file
do
   case $file in
   *.c)  cfile=$file
      ;;
   *.h)  hfile=$file
      ;;
   esac
done
if [ –z "$cfile" –o –z "$hfile" ]
then
   echo You must give me a .c file and echo a .h file, but one is missing.
   exit 1
fi
echo cc $cfile
cc $cfile
```

Figure 4
Build script with error checking

then the umask should be set in the script to a value that will prevent other students from accessing these temporary files.

The **run** script in Figure 5 runs a predetermined number of tests. The test number is passed as an argument to the program; presumably the main program was written by the instructor and uses this argument to decide which test to run. The main program in this example prints only when errors are encountered, hence the test for non-zero size of the output file.

```
#! /bin/sh
for T in 1 2 3
do
   echo Running test $T
   try_run a.out $T > out
   if [ –s out ]
   then
      exit $T
   fi
done
exit 0
```

Figure 5
Run script for several tests

The **run** script in Figure 6 also runs a predetermined number of tests. In this case, however, the build script created three executable programs, each linked with different versions of the instructor's routines. The executables were named "test.1", "test.2", and "test.3". Again, nothing is printed unless errors are detected. Note that the exit status of **try_run** is checked in this script. Along with the use of **try_log** this records more detailed failure information in the log file (e.g. cpu time limit exceeded, memory fault, etc.).

Figure 7 shows a run script that executes the student's program with several different input files. Each input file is accompanied by a description file that is printed if the test fails (see *The explain Script*, below), and by an output file that contains correct answers. The script runs until it runs out of input files or an error occurs.

```
#! /bin/sh
for T in 1 2 3
do
   echo Running test $T
   try_run cpu=1 test.$T > out
   status=$?
   if [ $status –ne 0 ]
   then
      try_log $status
      exit $T
   fi
   if [ –s out ]
   then
      exit $T
   fi
done
exit 0
```

Figure 6
Run script for several executables

2.5.4. The explain Script

The purpose of this script is to explain to the student the test that caused the project to fail. This might involve copying an input file to the terminal, copying a description file, and so forth. It might involve nothing at all if the instructor so chooses.

The test number that was returned by the run script is passed as the only argument. The exit status of this script has no significance.

It is quite possible for many projects to have identical **explain** scripts. Therefore, if no **explain** script is found in the project directory, the **explain** script (if there is one) in the script directory (or the try directory if no **script_dir** was given) will be executed instead.

The following script simply copies an input file to the terminal.

```
#! /bin/sh
echo Here is the input for test $1
cat $WD_BACKPATH/$PROJ_DIR/input.$1
```

The script in Figure 8 checks the high-order bit in the test number to determine whether or not to reveal the input file. The **run** script would have set this bit depending on how the student's project failed.

If the stuff printed by the **explain** script is long, the *more*(1) command should be used to prevent it from scrolling off of the screen before the student can read it. This is illustrated in the previous example. It is important for security reasons to pipe the data to be printed into *more* rather than giving filenames as arguments to *more* (see *Using More*, below for details). The **–d** flag was given to cause *more* to print a descriptive prompt when it pauses at the bottom of the screen. This is helpful to novice users who are unfamiliar with its operation.

```
#! /bin/sh
T=1
D=$WD_BACKPATH/$PROJ_DIR
while [ –r $D/test.$T ]
do
   echo Running test $T
   IN=$D/input.$T
   OUT=$D/output.$T
   try_run fsize=1k \
      a.out < $IN > ,x
   status=$?
   if [ $status –ne 0 ]
   then
      try_log $status
      exit $T
   fi
   if cmp –s ,x $OUT
   then
      true
   else
      exit $T
   fi
   T=`expr $T + 1`
done
exit 0
```

Figure 7
Run script using input and output files

2.5.5. The cleanup Script

This script is optional. If it exists, it is run after the **run** or **run_safe** script has finished (and after the **explain** script, if there is one, is finished). One argument is passed to this script: "0" indicates that the project could not be built or failed a test, and "1" indicates that the student successfully completed the project. If it is necessary to know the names of the files submitted by the student, the **FILES** environment variable may be used.

This script is most frequently used to perform cleanup tasks that could not be done at the end of the **run_safe** script. It can also be used to perform cleanup tasks unique to a particular project so that

```
#! /bin/sh
if [ $1 –gt 128 ]
then
   echo The cause of the failure can be determined from the output you
   echo have already seen.
   exit
fi
echo Here is the input for test $1
cat $WD_BACKPATH/$PROJ_DIR/input.$1 | more –d
```

Figure 8
Explain script

a common **run** script could be used. If the **save** option was given for a project in the **.tryrc** file, the scratch directory will not be cleaned out. The **cleanup** script can be used in this case to remove any unneeded files from the scratch directory so that only the good stuff is kept.

The exit status of the **cleanup** script is ignored unless the **cleanupstatus** option is given. In this case, a non-zero status causes **TRY** to log the failure and exit, and the **try_log** program may be used in the **cleanup** script to add a more detailed explanation to the log file.

Figure 9 shows a **cleanup** script that copies the student's files to an archive that can be perused later, for example, to grade the student's programming style.

2.5.6. The dump_log Script

This script is executed when the student uses the **–l** option in **TRY**. Its purpose is to print portions of the student's log file. It is run from within the scratch directory, although no student files are copied with **-l**. It is called with the name of the log file and the project code as arguments; the log file name is the pathname to the log file from the scratch directory. The instructor can print as much or as little information from the log file as he wants. The exit status of this script has no significance.

```
#! /bin/sh
ARCH=$WD_BACKPATH/arch_$PROJ_DIR
if [ ! -d $ARCH ]
then
   mkdir $ARCH
fi
rm *.o
tar cf $ARCH/$STUDENT *
compress $ARCH/$STUDENT
```

Figure 9
Cleanup script

It is likely that most projects will have identical **dump_log** scripts. Therefore, if no **dump_log** script is found in the project directory, the **dump_log** (if there is one) in the script directory (or the try directory if no **script_dir** was given) will be executed instead.

The script below simply prints the entire log file, regardless of the project code.

```
#! /bin/sh
cat $1 | more –d
```

The next script prints only those entries for the specified project. The → symbol in the *egrep* patterns below represents a tab. This is adequate so long as none of the project codes look like dates or times.

```
#! /bin/sh
egrep "→$2→" $1
```

The script in Figure 10 simply tells a student whether he has completed the specified project. The final example in Figure 11 shows a script that

```
#! /bin/sh
X=`egrep "→$2→.*→Completed" $1`
if [ –n "$X" ]
then
   echo Completed
else
   echo Not completed
fi
```

Figure 10
Completion status dump_log script

allows a student to examine his entire log file as well as the entries for a specific project. This is done with a dummy entry "all" in the **.tryrc** file. In this case, the log file is sorted first, which brings all of the entries for each project together before printing them.

2.5.7. The query Script

This optional script is intended to provide a mechanism by which the student can obtain information about a project. It is executed when the **–q** flag is given to **TRY**. The script is executed from the scratch directory like the rest of the scripts (although no student files are copied for queries). An instructor could write a query script that divulged information about the tests for a project, collected information from the student, gave hints for completing the project, and so forth. Queries are not recorded in the log file.

There is no **query_safe** script. Like the other unprotected scripts, the **query** script is executed with the instructor's user-id. Because of its general nature, one must be particularly vigilant when writing this script to avoid commands that might allow the student to execute arbitrary programs. For example, it is unsafe to invoke an editor from the script and allow the student to enter editing commands; the student could use the editor's shell escape to execute arbitrary commands in the instructor's account. **Try_run** can be used to execute any such dangerous program in the **query** script, so long as any files that need to be accessed by the program reside in (or have links in) the scratch directory.

2.6. Modifying the Environment

While **TRY** begins with the environment variables defined by the student, it modifies some of these values for security reasons (see *Security and The Environment*, below). The instructor is then given two opportunities to further modify the environment.

```
#! /bin/sh
if [ "$2" = all ]
then
   sort $1 | more -d
else
   egrep "→$2→" $1
fi
```

Figure 11
Dump_log script for all entries

First, environment variables may be set in the **.tryrc** file, as previously described. Second, the **init**, **build**, **build_safe**, **run**, and **run_safe** scripts may all create a file named **.ENVIRONMENT** containing environment variable definitions of the form

```
NAME=value
```

If such a file exists after one of these scripts finishes, **TRY** will add those variables to its environment so that their values will be available to the subsequent scripts. Should a variable which already has a value be set in this manner, the new value replaces the old one.

2.7. Status Messages

TRY informs the student when the build script fails, when a test in the run script fails, and when the project is successfully completed. The instructor can specify the text to be used in these messages with environment variables set in the **.tryrc** file. There is an environment variable for each of these messages. If it is explicitly empty, the message is not printed at all.

These messages are all used as the format string for a C language **printf.** Figure 12 shows the default messages, the names of the relevant environment variables, and the argument printed in each message. If the text supplied for a message does not contain a format code, the argument will not be printed. If it does contain a format code, the code must be compatible with the one shown in the table. You deserve what you get if your format code is not compatible, or if you give more than one.

2.8. News

TRY implements a capability to allow the instructor to provide students with news. If there is a file called **news** in the try directory with read permission for the owner, its contents are written to the student's terminal after the files have been copied. This file is intended to be used for general information for a class. There can also be a **news** or **news.***project-code* file in each of the project directories for specific information about that project. The former is used for projects with their own project directories, and the latter for projects which share a common project directory. If both exist, only the latter file is used. This news is only printed when the specific project is submitted.

A status file is kept in the student's home directory recording which news files have been seen. This is keyed on the date that news files

Env Variable	*Argument*	*Format*	*Default message*
TRY_SUCCESS	project code	%s	You have successfully completed project %s!
TRY_FAIL_BUILD	project code	%s	Couldn't build your project.
TRY_FAIL_RUN	test number	%d	Your project failed test %d.

Figure 12
TRY status messages

were modified, so any change to a news file will cause it to be shown to the student during their next run of **TRY**. If the **–n** flag is given to **TRY**, general news (if any) and news for the named project (if any) are printed regardless of whether the student has seen it before.

News is not printed when running in the background (**–b**) mode.

2.9. Errors

Errors that may occur when a student runs **TRY** either have to do with the student's files or with the instructor's scripts and files. There are also several "impossible" errors that should never occur, such as not being able to fork or create a pipe, etc. Errors that are the fault of the student are reported to the student. These include files that the student named which cannot be accessed, and incorrect instructor or project code arguments. Interrupts are also reported to the user.

When any other error occurs, **TRY** tells the student to see their instructor, and records a descriptive error message in the student's log file. This prevents students from seeing the gory details of the instructor's mistakes, and also allows the instructors to see the real error message rather than relying on what the student thought they saw but didn't write down and is no longer quite sure of.

The check option to **TRY** (described below) allows the instructor to verify that the required scripts exist and that the scripts and directories have the proper modes. Also, after setting up a project, it is a good idea to test your scripts by using **TRY** from a different account (e.g. if you have set up **TRY** in your grader account, test it from your faculty account). This often spots problems in scripts or with permissions of data files that might otherwise have caused trouble for the students.

3. Security

TRY runs as a set-user-id program in order to be able to access the instructor's files without requiring that the student have access to them. In this regard, the information in the try directory is as safe from students as the instructor cares to make it. The difficulty with security comes when the student's program is executed in the scratch directory. Presumably this program could do *anything*; for example, deleting files in the current and parent directories. A more sophisticated user might make a copy of **/bin/sh** in his own directory and set the mode so that it is set-user-id to the faculty account, or remove failure entries from his own log file, and so forth. The really careful student can arrange to leave no traces of his mischief.

For such things to happen, the student's program must be executed by the instructor's account. However, this is typically what happens with *submit*(1R), where the instructor runs shell scripts to execute the students' programs and save the output for later perusal.

These and other security issues were considered in the design of **TRY**. It is the author's belief that a moderately careful instructor is completely safe using **TRY**[2]. Neither people nor software are perfect, however, hence the assertion that it is better to use **TRY** in an account that does not have a lot of power than in a faculty account.

3.1. The Safe Scripts

The **build** and **run** scripts are executed with the instructor's user-id. This is acceptable for most commands that might be needed to build and test a project, and simplifies accessing files in the project directory and elsewhere. There are commands, however, that are risky to execute with the instructor's identity. Running *make*(1) with a student-supplied makefile, for example, allows the student to execute arbitrary commands with the instructor's identity in the instructor's account.

The **build_safe** and **run_safe** scripts are executed with a nonexistent user-id and group-id. Commands executed in these scripts will be able to access only those files that have public permission.

[2] But see *Using Make*, below.

Arbitrary commands from a student makefile will therefore be limited by the public permissions set in the try directory rather than by the instructor's permissions, and can therefore be rendered harmless.

To allow access to files in the project directory from the **build_safe** and **run_safe** scripts, the try directory and project directory must both have at least mode 111. The files that are accessed must also have the appropriate public permission set. It is recommended that only the required permission be given, and no more. 711 would be a typical mode for the try directory and for project directories. This would allow the safe scripts to access files in the directory, but would not allow malicious commands to list the directories or add or delete files in them.

Mode 711 for the try directory allows outsiders into the directory if they already know the names of files contained therein. To obtain complete protection from outsiders, the try directory should be created in a directory whose mode is 700. The required public mode for the try directory also does not prevent a malicious student program from accessing files in the directory whose names are known in advance. Such access is prevented by the permissions on the files themselves. The absence of write permission on the try directory and project directories prevents the malicious student from creating new files or deleting in them.

The scratch directories require mode 777 as the nonexistent user will be creating and possibly deleting files in them. **TRY** creates them with this mode and makes sure they still have the proper mode before each use. Files copied into the scratch directory will be owned by the nonexistent user, with their group and public permissions set the same as the owner permissions on the student's copy of the file.

Figure 13 shows a script that tests a student's makefile. The **run_safe** script was used because of the need to run *make*.

This script has several noteworthy features. First, **try_run** is used to execute *make*. This guards against infinite loops in the commands the student might execute. The **–p** option is used to allow access to commands in system directories; this is safe because the script is executed with the nonexistent user-id.

If the shell variable **TRY_SAVE_ANS** is set to any value, the script will copy the student's output files and use them as the correct answers. The instructor could use this feature with his own makefile to easily create the correct output files. It is important that this variable either begin with "TRY_" or be explicitly set empty to guard against the possibility of the student having an environment variable with the same name. This script does both.

```
#! /bin/sh
D=$WD_BACKPATH/$PROJ_DIR
TRY_SAVE_ANS=
T=1
while [ –r $D/test.$T ]
do
   echo Running test $T
   touch `cat $D/touch.$T`
   try_run –p cpu=20 \
      /bin/make > ,x
   status=$?
   if [ $status –ne 0 ]
   then
      try_log $status
      exit $T
   fi
   awk –f $D/awk1 < ,x |
      try_deblank –b –r > ,$$
   if [ –n "$TRY_SAVE_ANS" ]
   then
      cp ,$$ $D/output.$T
   fi
   if cmp –s ,$$ $D/output.$T
   then
      true
   else
      exit $T
   fi
   T=`expr $T + 1`
done
exit 0
```

Figure 13
Run_safe script

Note that between *make*s, various files are touched to see how the student's makefile reacts to the changes. Also, the *make* output is massaged by **try_deblank** and an *awk* program before being compared to the solutions; this takes care of the possibility that the student's makefile compiles source files in a different order than in the instructor's makefile.

3.2. Security and The Environment

The environment is another source of security problems. The student is executing **TRY**, so it is the

student's environment that is passed by **TRY** to everything it executes. For this reason, the environment is laundered as described below before being passed to any of the instructor's scripts. Figure 14 summarizes the environment variables that are added or changed.

Experienced shell script writers often use the **HOME** environment variable. Were it not changed, this would be the student's home directory. Therefore, **HOME** is set to the home directory of the instructor's account. The student's home directory is kept in the new variable **STUDENT_HOME** .

A new variable called **PROJ_DIR** is added to the environment. Its value is the name of the project directory. This is useful in shell scripts that are used by more than one project. The variables **INST** and **PROJ** are also created; their values are the instructor's account and the project code, respectively.

A common strategy to circumvent security provisions is to create one's own versions of system programs (e.g. cc, ls, etc.) in one's own bin directory, and set the **PATH** environment variable to search that directory before the system directories. If the instructor uses any of these command in his scripts, the student's version of the program will be executed, presumably with the instructor's identity. To avoid this, **TRY** sets **PATH** to

```
/usr/local/bin:/usr/ucb:/bin:/usr/bin
```

This ensures that only the standard directories are searched for programs used in shell scripts.

The current directory is absent from **PATH** to prevent the student from passing his version of a system program in as a file and executing it from the scratch directory. This also makes it less likely that the instructor will accidentally execute the student's program directly in a script rather than with **try_run** (see below), which would be a security breach.

If the instructor wishes to use programs in his own **bin** directory in **TRY** scripts, **PATH** can be changed in the appropriate scripts as in the following example:

```
PATH=$HOME/bin:$PATH
```

One should *not* put the current directory in the **PATH** as that could cause scripts or programs submitted by the student to be run instead of standard programs.

An environment variables **FILES** is created; its value is the list of file names submitted by the student.

If the student used the **–b** option to run in the background, the environment variable **BACKGROUND** will be set. The instructor may test this in scripts to decide, for example, whether or not to use *more*(1) (which is interactive) to print to the screen.

The environment variable **SHELL** is changed to be "/bin/echo" in order to prevent users from gaining access to the shell through *more*. (Also, see *Using More*, below.) Unfortunately, many other programs also examine this variable, and their

Variable	*Set to*
BACKGROUND	"yes" if the student gave the **–b** option, else not set.
FILES	a list of the file names given by the student.
HOME	the name of the instructor's home directory.
INST	the name of the instructor's account.
LOG_DIR	the directory containing the log files (relative to the try directory)
PATH	the standard value "/usr/local/bin:/usr/ucb:/bin:/usr/bin"
PROJ	the project name given by the student.
PROJ_DIR	the name of the project directory.
SCRIPT_DIR	the directory containing the common scripts (relative to the try directory)
SHELL	the bogus value "/bin/echo"
SIGNAL	the number of the signal that occurred
STUDENT	the student's account name
STUDENT_HOME	the name of the student's home directory.
TRY_STARTTIME	the time of day when **TRY** was started.
WD_BACKPATH	the pathname to get back to the try directory from the scratch directories
WD_DIR	the directory containing the scratch directories (relative to the try directory)

Figure 14
Environment variables

operation will also be affected by its change. Those that use it to do shell escapes will no longer be able to do so, which is just what is desired. Programs that use it for other reasons may not work as expected in **TRY** scripts.

The environment variable **STUDENT** is set to the student's account name.

Setting any of these environment variables from the **.tryrc** file supercedes the actions described above. This gives the instructor a way, for example, to set his own **PATH** variable without having to remember to put it in each and every script.

All variables in the student's environment that begin with "TRY_" are deleted. This allows the instructor to use variables with this prefix without having to worry about interference from the student.

3.3. Using More

Throughout this document, the *more*(1) command has been suggested as the way to present text to the user, as it prevents long output from scrolling off of the top of the screen before the user has had a chance to read it. Unfortunately, *more* itself is a source of two security problems.

The first problem is the shell escape mechanism, which is effectively disabled when **TRY** changes the **SHELL** environment variable (see above). The second problem is the ability to invoke an editor on the current file being printed. From the editor, the student might be able to obtain a shell (depending on whether the editor invoked examines the **SHELL** variable). This can be prevented by using *more* only in a pipeline, as in the following example:

```
cat errors | more –d
```

More will not start an editor when its input is from a pipe, thus avoiding the security problem.

More really ought to have a "secure mode" flag that would prevent it from starting editors or invoking shells in sensitive situations. Until it does, one must be careful when writing scripts to never invoke *more* directly on a file. The other alternative is to simply not use it, or to write one's own.

3.4. Using Make

The need to run *make* from a safe script when students submit their own makefiles has already been discussed. However, the safe scripts cannot prevent this problem: if the student starts a shell from his makefile, then he can *cd* to other scratch directories and try to make copies of files submitted by other students. This is hard to prevent because of the liberal permissions needed on the scratch directories in order to allow processes with the nonexistent user-id to work in them. However, the default is to delete everything in the scratch directory at the end of the submission, so the problem reduces to one of timing: copies may be obtained only when the other student is actually running **TRY**, and then only when the students' own makefiles are used. Students can be very tenacious, however.

The safest strategy is not to take student makefiles. But if you must, consider filtering them to look for invocations of a shell.

4. Directory and File Modes

The table below specifies the minimum modes needed for things in the try directory.

What	*Mode*
try directory	111
project directory †	111
wd_dir ‡	100
script_dir ‡	100
log_dir ‡	100
news files	400
init script	500
build script	500
build_safe script	555
run script	500
run_safe script	555
explain script	500
cleanup script	500
dump_log script	500
query script	500

† If the pathname for the project directory contains subdirectories, each component of the name must also have at least this permission.

‡ These all default to the try directory. If a pathname is given for any of these, each component of the name must have at least this permission.

Files in the project directory that are accessed from the **build_safe** or **run_safe** scripts must have public permission appropriate for the type of access. Usually this is read permission, meaning that they will require at least mode 444.

4.1. The Check Option

The **–c** option to **TRY** causes it to check that the various directories and scripts are present and have the proper mode. Improper modes will be

changed. *Note:* only the scripts and directories in the table above are checked with this option. Files accessed from within the instructor's scripts are not checked for proper permissions.

This may only be used by the instructor. The instructor account and project code must be given, but no files are required for this option. Sample output from the check option is shown in Figure 15.

5. Signals

Signals that are likely to be generated by the user include **SIGINT** (the delete or rubout key), **SIGHUP** (modem hangup), **SIGQUIT** (keyboard quit signal), and **SIGTERM** (from the *kill*(1) program). **TRY** always honors these signals while the student's files are being copied (for when the student sees that he has given the wrong file), and always ignores them after the **run** script has finished (to prevent students from being able to interrupt the **cleanup** or **explain** scripts). The way in which **TRY** handles these signals while the project is being built and tested depends on options given by the instructor, whose effects are explained below.

All other signals always cause **TRY** to terminate abnormally, with the signal name reported in the log file if possible. These signals would ordinarily indicate the presence of a bug in the **TRY** program (e.g. a memory fault), however, the student can always send such a signal with the *kill* program.

5.1. Signal Handling Options

By default, the the four user signals are handled the same as the other signals. Their receipt causes **TRY** to terminate abnormally, reporting the signal name in the log file. Shell scripts will also receive these signals, and instructors are free to implement any desired signal handling actions in scripts with the **trap** or **onintr** shell statements.

Two of the options that can be given in the **.tryrc** file modify the way the four user signals are handled. The **ignore** option causes them to be ignored, and causes all shell scripts to ignore them as well. The instructor may wish to explicitly handle other signals in his scripts.

The **post** option causes **TRY** to ignore the four user signals, but does not arrange for shell scripts to ignore them. The instructor can handle them if desired with the **trap** or **onintr** shell statements. In addition, when **TRY** receives a signal, it adds the variable **SIGNAL** to the environment; its value is the number of the signal received (see *signal*(2) or *sigvec*(2)). If desired, the instructor can test for this variable in his shell scripts and take appropriate action if it is found.

Giving both the **post** and **ignore** options has the same effect as **ignore**, except that the **SIGNAL** environment variable is posted when a user signal is received.

One must be aware that these options only change the default signal handling actions. If a program explicitly arranges to catch signals, the ignored signals will no longer be ignored. A few programs do this; the C compiler is a notable example. Even with the **ignore** option set, the user can interrupt the C compiler, which will then return a failing exit status so at least the build script will behave reasonably. If programs used in the run script arrange to catch their signals, it could cause more of a problem. It is unfortunate that there is nothing that can be done in **TRY** to remedy this.

6. Utility Programs

There are several utility programs that were developed for **TRY**; these are described below.

```
$ try –c kar 4–1
try: Looking in directory "class/309/labs":
try: "make/build_safe" and "make/build" both exist, the former will be used.
try: "make/build_safe" exists but has mode 600.  It needs at least mode 555.
try: I changed it to 755 for you.
try: "make/run_safe" exists but has mode 600.  It needs at least mode 555.
try: I changed it to 755 for you.
try: "make/explain" exists but has mode 0.  It needs at least mode 500.
try: I changed it to 500 for you.
$
```

Figure 15
Sample output from the check option

6.1. Try_run

This program is used to run a student's executable program in a controlled manner. It limits the resources that can be used by the program to catch infinite loops and prevent huge files from being created. It prevents the program from accessing any files outside of the current directory to protect the instructor account from malicious actions that a student program might take. Its exit status can be tested in the script to determine how the student's program finished.

See *try*(1R) for complete information.

6.2. Try_log

This program inserts a message into the log file, and may be used in build and run scripts. If one command line argument is given and it is numeric, it is assumed to be an exit status from **try_run** and a message explaining the status is added to the log file. Otherwise, the text of the argument(s) is taken as the message and inserted into the log file. Newlines found in the message are removed. Messages that are too long (e.g. several hundred characters) should be avoided as they may break programs that read the log files. **try_log** may be used before successful as well as unsuccessful termination of a script, but it should be executed only once.

Several of the previous examples illustrated the use of **try_log** to report error status from **try_run** and to add other messages to the log file. Figure 16 shows a build script that uses **try_log** to record whether a failure was due to compilation or linking errors. This might be taken into account when grading the project.

See **try_log** for complete information.

6.3. Try_deblank

This program performs commonly used text manipulations, and is often used to filter the output of a student's program before checking it for correctness. Options available include removing all leading and trailing blanks, adding one leading and one trailing blank, reducing runs of blanks to a single blank, removing blank lines, and printing the input one word per line. Tabs may also be treated as blanks in the above transformations.

See **try_deblank** for complete information.

7. Detecting Cheating

TRY deletes the contents of the scratch directory when it is finished. Instructors might argue that this makes it more difficult to detect cheating by removing the evidence. The instructor is always free to make copies of whatever files he wishes in any location outside of the scratch directory, as illustrated in the sample **cleanup** script given above. To help detect cheating, the instructor would also wish to run some sort of program to help find duplication among different students' files. The example below, executed at the end of a run script, records in the log file a checksum for each of the student's files.

```
try_log `sum $FILES`
```

```
#! /bin/sh
echo cc –c $*
cc –c $*
if [ $? –ne 0 ]
then
   try_log cc
   exit 1
fi
D=$WD_BACKPATH/$PROJ_DIR
ln $D/print–o print.o
echo cc *.o
cc *.o
if [ $? –ne 0 ]
then
   try_log ld
   exit 1
fi
exit 0
```

Figure 16
Build script using try_log

8. Where Implemented

Currently, **TRY** is installed on all of the Sun servers (solaris).

9. Appendices

The first appendix summarizes the purposes and requirements of all of the shell scripts written by the instructor. Manual pages for these programs are included next. The shorter **TRY** manual page is intended for distribution to students, who need to know how to run the program but do not need to know the details of how project tests are set up.

Script Name	***Req/ Opt***	***Min Mode***	***Arguments Passed***	***Exit Code***
init	opt	500	student's file names	0=success, 1=failed (4)
build	(1)	500	student's file names	0=success, 1=failed
build_safe	(1)	555	student's file names	0=success, 1=failed
run	(2)	500	student's file names	0=successful, else return number of the failed test
run_safe	(2)	555	student's file names	0=successful, else return number of the failed test
explain	(3)	500	number returned by run script	(none)
cleanup	opt	500	0=project failed 1=project succeeded	0=success, 1=failed (5)
dump_log	(3)	500	name of log file, project code	(none)
query	opt	500	(none)	(none)

Notes:

Scripts that require a minimum mode of 555 are executed by a nonexistent user-id. Other scripts are executed by the instructor's user-id.

(1) Either the **build** or the **build_safe** script is required. If both are present, the **build_safe** script will be used.

(2) Either the **run** or the **run_safe** script is required. If both are present, the **run_safe** script will be used.

(3) If there is no **explain** script in the project directory, the **explain** script (if there is one) in the script directory will be used. The same is true for the **dump_log** script.

(4) The status of the **init** script is relevant only when the **initstatus** option is given.

(5) The status of the **cleanup** script is relevant only when the **cleanupstatus** option is given.

Appendix 1
Summary of Shell Scripts