Test Environment Toolkit

TETware Knowledge Base
Revision 1.1
TET3-KB-1.1

Released: 19th November 1999

The Open Group

The information contained within this document is subject to change without notice.

This document is produced by UniSoft Ltd. at:

150 Minories
LONDON
EC3N 1LS
United Kingdom

CONTENTS

PERMUTED INDEX

1. Introduction

1.1 Preface

The TETware Knowledge Base contains articles which describe how to use TETware to solve particular testing problems. Many of the articles contain information that has previously been provided in response to specific questions from TETware users.

TETware is implemented on UNIX operating systems and also on the Windows NT and Windows 95 operating systems. It includes all of the functionality of the Test Environment Toolkit Release 1.10 (TET), the Distributed Test Environment Toolkit Version 2 Release 2.3 (dTET2) and the Extended Test Environment Toolkit Release 1.10.3 (ETET), together with a number of new features.

Throughout this document, the Windows NT and Windows 95 operating systems are referred to collectively as **Win32 systems**. The individual system names are only used when it is necessary to distinguish between them.

1.2 Audience

This document is intended to be read by software engineers are already familiar with TETware. People who are new to TETware should first refer to the documents described in the section entitled ‘‘Related documents’’ later in this chapter.

1.3 Conventions used in this document

The following typographic conventions are used throughout this document:

- `Courier font` is used for function and program names, literals and file names. Examples and computer-generated output are also presented in this font.
- The names of variables are presented in *italic font*. You should substitute the variable’s value when typing a command that contains a word in this font.
- **Bold font** is used for headings and for emphasis.

Long lines in some examples and computer-generated output have been folded at a `\` character for formatting purposes. If you type such an example, you should type it in all on one line and omit the `\` character.

Unless stated to the contrary, shell script examples in this document assume the use of a Bourne, Korn or POSIX shell on UNIX systems or, where applicable, the MKS Korn Shell on Win32 systems.

1.4 Related documents

Refer to the following documents for additional information about TETware:

- *Test Environment Toolkit: TETware Installation Guide*
  There is one version of this document for each operating system family on which TETware is implemented.
- *Test Environment Toolkit: TETware Programmers Guide*

- *Test Environment Toolkit: TETware User Guide*

In addition, the TETware Release Notes contain important information about how to install and use TETware. You should read the release notes thoroughly before attempting to install and use each new release of TETware.

2. Running `tcc` and `tccd` with different user IDs in Distributed TETware

This information is not applicable to Win32 systems.

Normally, `tcc` and `tccd` are run with the same user ID on the local system. This should be the same as the owner of the files below the test suite root directory on the local system. This arrangement is desirable because in Distributed TETware both `tcc` and `tccd` may create files in the test suite directory hierarchy. By default, `tccd` runs with the ID of the user `tet` and so it is appropriate for `tcc` to run with the ID of the user `tet` as well.

Sometimes it is necessary to run `tcc` with a different user ID from the one used by `tccd`. These instructions assume that you will run `tccd` as the user `tet` and the people who invoke `tcc` will use their own login IDs.

You should perform the following operations:

1. Create a group called `tet` in the `/etc/group` file. Arrange for the supplementary group list of all the users who will use Distributed TETware to include the `tet` group. On most systems, you do this by listing all the login names in the last field of the `tet` entry in `/etc/group`.

   For example:

   ```
   tet::102:tet,john,paul,george,ringo
   ```

2. In the `/etc/passwd` file, make sure that the `tet` user has the `tet` group as the primary group (this is shown in the fourth field in the password entry).

   For example:

   ```
   tet:x:106:102:TETware user:/home/tet:/bin/ksh
   ```

   Note that the 102 is the same as the 102 in the `tet` group entry.

3. Run `tccd` with a `umask` of 2 (by using the `-m` command-line option). This will cause files created by `tccd` and the test cases and tools that `tccd` executes to be writable by group as well as by user.

   For example, you might put the following entry in `/etc/inetd.conf:`

   ```
   tcc  stream  tcp  nowait  tet  /home/tet/bin/in.tccd  in.tccd -m2
   ```

4. Make sure that each user who will work with Distributed TETware has a `umask` of 2 (or less restrictive). You can do this by putting the line:

   ```
   umask 2
   ```

   in each user's `.profile`.

5. Make sure that all the directories below `$TET_ROOT` belong to the group `tet` and are writable by group. You can do this by executing the following commands:

   ```
   cd $TET_ROOT
   find . -type d ! -group tet -print | xargs chgrp tet
   find . -type d -print | xargs chmod g+w
   ```

   Note that this assumes that you have not yet built any of the test cases. If you have, you will need to find all the build products and change them to group `tet` and mode 664 as well.

6. Make sure that files and directories created below `$TET_ROOT` inherit the group of the directory in which they are created. On some systems this is the default behaviour, whereas on others you must turn on each directory's set-GID bit in order to enable this behaviour. If you need to turn on the set-GID bits you can do this by executing the following commands:

```
cd $TET_ROOT
find . -type d -print | xargs chmod g+s
```

Points to note:

a. These instructions assume that the test case files are readable by owner and group (at least). If they're not, the following command will fix this:

```
cd $TET_ROOT
find . -type f -print | xargs chmod ug+r
```

b. The instruction to set the set-GID bit on directories is to ensure that newly-created files and directories inherit their group ID from the parent directory. If this is your system's default behaviour, you don't need to set the set-GID bit in order to achieve this.

c. These instructions assume that your system keeps password and group information in `/etc/passwd` and `/etc/group`. If your system uses NIS for its password and group databases you should ask your system administrator what to do.

d. On some systems it is not advisable to edit the password and group files directly. Instead you should use the appropriate system administration commands to add and/or change entries in these files.

See also

- ''The Test Case Controller daemon `tccd`'' in the TETware User Guide.
- The `tccd` manual page in the TETware User Guide.

3. How does TETware handle the `tet_xres` temporary results file?

If the test case is a distributed API-conforming test case, then the API must use `tetxresd` (the execution results daemon) for the journal. Otherwise, the API may either use `tetxresd` or generate its own `tet_xres` file (but not both!).

In this context:

1. **distributed** means that either
   a. the `:distributed:` scenario directive is used; or
   b. the `:remote:` scenario directive is used and system 0 is included in the system list
2. **API-conforming** means that either
   a. `TET_API_COMPLIANT` is true; or
   b. `TET_API_COMPLIANT` is unset and `TET_OUTPUT_CAPTURE` is false

The TETware `tcc` looks for a `tet_xres` file in the test case execution directory when an API which does not support distributed testing is used. (Currently: all the APIs except the Distributed C and C++ APIs.)

All the journal processing is performed by `tcc`. `tccd` is only used when a `tet_xres` file is on a remote system, and then only to copy it to the local system for processing by `tcc`.

Implementation details

Journal processing is performed by one of the functions in `tcc`'s execution engine. This function is called `tcs_journal()` in file `tcc/proctc.c`. A large comment in the code describes the strategy that is used; it reads as follows:

```
/*
** for an API-conforming test case:
**      if there is a journal at this level:
**              if the tool used XRESD:
**                      process the XRESD file;
**              otherwise:
**                      the tool should have used tet_xres;
**                      if there are child proctabs:
**                              process each child's tet_xres file;
**                      otherwise:
**                              process this level's tet_xres file;
**      otherwise:
**              for each child journal:
**                      if the tool used XRESD:
**                              process the XRESD file;
**                      otherwise:
**                              the tool should have used tet_xres;
**                              process the tet_xres file;
** otherwise:
**      if there is a journal at this level:
**              if PRF_AUTORESULT is set (i.e., non-API in EXEC mode):
**                      generate a TP result from pr_exitcode;
```

```
**      otherwise:
**              for each child journal:
**                      if PRF_AUTORESULT is set:
**                              generate a TP result from pr_exitcode;
**
*/
```

Note that the words ‘‘test case’’ and ‘‘tool’’ are used interchangeably — `tcc` processes test cases and tools in the same way.

In the simple case of an API-conforming test case running on the local system, `tcs_journal()` calls `jnlproc_api()` which is in file `tcc/jnlproc.c`.

A combined execution results file is always opened (by `tetxresd`) for an API-conforming test case before the test case is processed. However, this file will remain empty unless an API that supports distributed testing is used. The code in `jnlproc_api()` looks at the combined execution results file to see if there is anything there, and uses it if there is. Otherwise, `jnlproc_api()` assumes that another API has been used and looks for the `tet_xres` file instead. In the simple case, this processing is performed by a call to `jp_tetxres()`, which opens the file. Once the `tet_xres` file is opened, a call to `jp_xres()` transfers the contents of the file to the journal.

See also

- ‘‘Test reporting and journaling’’ in the TETware Programmers Guide.
- ‘‘Simple TETware architecture diagrams’’ and ‘‘The Execution Results Daemon `tetxresd`’’ in the TETware User Guide.

4. How does `tcc` locate the build tool?

`tcc` locates the build tool by using the `PATH` environment variable in the normal way. When `tcc` processes a test case in build mode it first changes to the test case source directory. Then `tcc` executes the build tool exactly as specified by the `TET_BUILD_TOOL` variable in the build mode configuration file.

Sometimes a test suite is organised so that the tools are located in a `bin` directory below the test suite root directory. `tcc` does not search this location automatically so, if you want to organise your test suite in this way, you should include this directory in your `PATH`. Alternatively, you can specify the location of the build tool relative to the test case source directory.

For example, in the `SHELL-API` test suite that is part of the `contrib` distribution, the build tool is *tet-root*`/contrib/SHELL-API/bin/buildtool` and the test cases live in directories below *tet-root*`/contrib/SHELL-API/ts`. The build mode configuration file contains the line

```
TET_BUILD_TOOL=buildtool
```

and the test case setup script prepends *tet-root*`/contrib/SHELL-API/bin` to the `PATH` environment variable. Thus `tcc` is able to find the build tool by using `PATH`. The same result could have been achieved without the need to modify `PATH` by specifying the location of the build tool relative to each test case source directory, thus:

```
TET_BUILD_TOOL=../../bin/buildtool
```

(Note that when this is done it is necessary for all the test case source directories to be on the same level in the directory hierarchy.)

Distributed TETware

In Distributed TETware the build tool is located as described above on each system. On a remote system the value of `PATH` that is used is the one in `tccd`'s environment on that system. On UNIX systems where `tccd` is started on demand from an entry in `/etc/inetd.conf`, the default value of `PATH` that `tccd` inherits from `inetd` is usually fairly minimal; for example: `PATH=:/bin:/usr/bin`. If necessary you can specify a different value of `PATH` by using the `-e` option to `tccd`. For example, on a SVR4 system where the compilation tools live in `/usr/ccs/bin`, you might put the following entry in `/etc/inetd.conf`:

```
tcc     stream  tcp     nowait  tet     tet-root/bin/in.tccd \
        in.tccd -e PATH=:/usr/bin:/usr/ccs/bin
```

Win32 systems

On Windows NT the o/s defines an environment variable called `Path` and the MKS shell defines a variable called `PATH`.

The `getenv()` function in the Microsoft C runtime support library is case-insensitive, so when `tcc` locates the build tool, it is possible that either the value of `PATH` or `Path` may be used. Therefore, if you change the value of `PATH` from the MKS shell, you should be sure to change the value of `Path` as well.

If you use a Shell script build tool on a Win32 system, its name must have a `.ksh` suffix. Alternatively you can say:

```
TET_BUILD_TOOL=sh
TET_BUILD_FILE=mybuildtool
```

in the execute mode configuration file.

A shell script should not start with a `#!` line since the MKS shell tries to interpret this and probably won't do what you expect. In any event, using `#!` is always non-portable on Win32 systems since the location of files can vary from system to system.

Other tools

All the information in this article also applies to the prebuild tool, the build fail tool and the clean tool.

The exec tool is processed slightly differently by `tcc` in that it might be executed in the test case source directory, in a location below the alternate execution directory, or in a location below the temporary directory, depending on the settings of certain configuration and environment variables. Therefore it is best not specify a `TET_EXEC_TOOL` with a relative path name because when the execution directory is changed the tool will no longer be found.

See also

- ''Build mode processing'' and ''Configuration variables which modify TETware's operation'' in the TETware Programmers Guide.
- The `tccd` manual page in the TETware User Guide.

5. Relationship between `TET_SUITE_ROOT` and `TET_TSROOT`

The TETware `tcc` provides support for the `TET_SUITE_ROOT` environment variable and the `TET_TSROOT` distributed configuration variable. The `TET_SUITE_ROOT` functionality is derived from ETET, whereas the `TET_TSROOT` functionality is derived from dTET2. Both these variables are used by `tcc` to locate the **test suite root** directory, but the meanings of the variables are not the same. In summary, `TET_TSROOT` refers to the test suite root directory itself, whereas `TET_SUITE_ROOT` refers to some path prefix of the test suite root directory (usually the parent directory).

Soon after `tcc` starts up it determines the location of the test suite root directory. Normally, `tcc` locates the test suite root directory below the **tet root** directory. However, you can use the `TET_SUITE_ROOT` environment variable to instruct `tcc` to look for the test suite root directory below a different location.

`TET_SUITE_ROOT` is also one of the communication variables. `tcc` puts this variable in the environment when it executes a test case or tool. If you don't specify a `TET_SUITE_ROOT` environment variable, `tcc` supplies one with a default value which is the same as the value of the `TET_ROOT` environment variable.

In Distributed TETware, `TET_SUITE_ROOT` is only used on the local system. The location of the test suite root directory on each remote system is specified explicitly by a `TET_REM_`*nnn*`_TET_TSROOT` distributed configuration variable in the `tetdist.cfg` file.

Since the `TET_SUITE_ROOT` feature is derived from ETET, remote and distributed test cases don't normally access this environment variable. So, by default, `tcc` puts an empty `TET_SUITE_ROOT` variable in the environment when it executes a test case or tool on a remote system.

However, in order to enable ETET test cases that expect to read `TET_SUITE_ROOT` from the environment to be processed on remote systems, it is possible to specify a value for `TET_REM_`*nnn*`_TET_SUITE_ROOT` in the distributed configuration file. When this is done, `tcc` assigns the specified value to the `TET_SUITE_ROOT` environment variable before processing the test case or tool on the remote system.

Note that the `TET_REM_`*nnn*`_TET_SUITE_ROOT` distributed configuration variable is only supported in order to provide backwards compatibility for ETET test cases on remote systems. It is not used by `tcc` itself since the location of the test suite root directory on a remote system must always be specified by the `TET_REM_`*nnn*`_TET_TSROOT` distributed configuration variable.

See also

- ‘‘Directory structure’’, ‘‘Communication variables’’ and ‘‘Distributed configuration variables used by Distributed TETware’’ in the TETware Programmers Guide.
- ‘‘Environment variables’’ in the TETware User Guide.
- The `tcc` manual page in the TETware User Guide.

6. Error message: `tcc/tcp: unknown service`

Question

When I run the Distributed version of `tcc` it prints the message:

```
tcc (tccdport.c, 80): tcc/tcp: unknown service: Bad file number
tcc (dtcc.c, 217): can't log on to TCCD on system 0
```

Answer

This message means that `tcc` can't find the entry for the `tcc` service in the services database on your system.

The services database associates service names with well-known port numbers. On UNIX systems it might be a file (usually `/etc/services`) or it might be an NIS database. On Windows NT systems it usually resides in the file `c:/winnt/system32/drivers/etc/services`.

You should find out where the services database is on your system and add an entry to it which describes the **tcc** service.

See also

- ''Services database entry'' in the TETware Installation Guide for your system type.

7. How to create a new process using `tet_spawn()` and `tet_wait()`

Question

Can you send me an example of how to use `tet_spawn()` and `tet_wait()`.

Answer

Here is a trivial test case that uses `tet_spawn()`. The source file is called `tc16.c`. You should compile it, then link with `tcm.o` and `libapi.a` in the usual way. (Or `tcm.obj` and `libapi.lib` on Win32 systems.)

```
#include <stdlib.h>
#ifndef _WIN32
#  include <sys/wait.h>
#endif
#include "tet_api.h"

void (*tet_startup)() = TET_NULLFP, (*tet_cleanup)() = TET_NULLFP;
void tp1();

struct tet_testlist tet_testlist[] = { {tp1, 1}, {TET_NULLFP, 0} };

#ifndef _WIN32
extern char **environ;
#endif

void tp1()
{
        pid_t pid;
        int status;
        static char *argv[] = {
                "./tc16child",
                "an-argument-string",
                (char *) 0
        };

        tet_infoline("this is tc16 parent");

        if ((pid = tet_spawn(*argv, argv, environ)) == -1) {
                tet_printf("tet_spawn(%s) failed: tet_errno = %d",
                        *argv, tet_errno);
                tet_result(TET_UNRESOLVED);
                return;
        }

        status = 0;
        if (tet_wait(pid, &status) == -1) {
                tet_printf("tet_wait(%ld) failed: tet_errno = %d",
                        (long) pid, tet_errno);
                tet_result(TET_UNRESOLVED);
                return;
        }
```

```
#ifdef _WIN32
        if (status != 0) {
                tet_infoline("child process returned unexpected exit status");
                tet_printf("expected exit status 0, observed %d", status);
                tet_result(TET_FAIL);
        }
        else
                tet_infoline("child exit status = 0");
#else
        if (WIFEXITED(status)) {
                if (WEXITSTATUS(status) != 0) {
                        tet_infoline("child process returned unexpected \
                                exit status");
                        tet_printf("expected exit status 0, observed %d",
                                WEXITSTATUS(status));
                        tet_result(TET_FAIL);
                }
                else
                        tet_infoline("child exit status = 0");
        }
        else if (WIFSIGNALED(status)) {
                tet_printf("child process terminated with signal %d",
                        WTERMSIG(status));
                tet_result(TET_UNRESOLVED);
        }
        else if (WIFSTOPPED(status)) {
                tet_printf("child process stopped by signal %d",
                        WSTOPSIG(status));
                tet_result(TET_UNRESOLVED);
        }
        else {
                tet_printf("can't decode child process exit status (%#x)",
                        status);
                tet_result(TET_UNRESOLVED);
        }
#endif

}
```

This is the child process that is launched by the call to `tet_spawn()` in `tc16.c`. The source file is called `tc16child.c`. You should compile it, then link with `tcmchild.o` and `libapi.a`. (Or `tcmchild.obj` and `libapi.lib` on Win32 systems.)

```
#include "tet_api.h"

int tet_main(argc, argv)
int argc;
char **argv;
{
        tet_infoline("this is tc16 child");
        if (argc > 1) {
                tet_printf("argument is \"%s\"", argv[1]);
                tet_result(TET_PASS);
        }
        else {
```

```
                tet_infoline("no arguments received");
                tet_result(TET_UNRESOLVED);
        }
        return(0);
}
```

Here is the journal that is generated when `tc16` is run on a Win32 system:

```
0|3.2-lite 23:19:40 19970714|User: unknown TCC Start, \
        Command line: tcc -epl /ts/tc16
5|Windows_95 TEXEL 4 0 586|System Information
20|c:/tet3/suite/tetexec.cfg 1|Config Start
30||TET_EXEC_IN_PLACE=false
30||TET_API_COMPLIANT=True
30||TET_PASS_TC_NAME=False
30||TET_VERSION=3.2-lite
40||Config End
10|0 /ts/tc16 23:19:40|TC Start, scenario ref 1-0
15|0 3.2-lite 1|TCM Start
400|0 1 1 23:19:42|IC Start
200|0 1 23:19:42|TP Start
520|0 1 0004883443 1 1|this is tc16 parent
520|0 1 0001218837 2 1|this is tc16 child
520|0 1 0001218837 2 2|argument is "an-argument-string"
520|0 1 0004883443 3 1|child exit status = 0
220|0 1 0 23:19:42|PASS
410|0 1 1 23:19:42|IC End
80|0 0 23:19:43|TC End, scenario ref 1-0
900|23:19:43|TCC End
```

See also

- ‘‘C language binding’’ and ‘‘Generating and executing processes’’ in the TETware Programmers Guide.

8. Relationship between `TET_OUTPUT_CAPTURE` and `TET_API_COMPLIANT`

Question

When I execute a test case I don't get any results in the journal.

Here is the start of the journal file:

```
0|3.2-lite 08:13:20 19970813|User: unknown TCC Start, \
        Command line: tcc -ep
5|Windows_95 WS1 4 0 586|System Information
20|c:/Tet3.2/suite/tetexec.cfg 1|Config Start
30||TET_OUTPUT_CAPTURE=true
30||TET_RESCODES_FILE=tet_code
30||TET_EXEC_IN_PLACE=true
30||TET_API_COMPLIANT=False
30||TET_PASS_TC_NAME=True
30||TET_VERSION=3.2-lite
40||Config End
700|3|Repeat Start, scenario ref 3-0
700|4|Repeat Start, scenario ref 6-0
10|0 /ts/tc1/tc1 08:13:20|TC Start, scenario ref 9-0
15|0 3.2-lite 1|TCM Start (auto-generated by TCC)
400|0 1 1 08:13:20|IC Start (auto-generated by TCC)
200|0 1 08:13:20|TP Start (auto-generated by TCC)
220|0 1 0 08:13:36|PASS (auto-generated by TCC)
410|0 1 1 08:13:36|IC End (auto-generated by TCC)
80|0 0 08:13:36|TC End, scenario ref 9-0
```

Answer

This journal contains a couple of clues as to what is going wrong:

1. The part that reports the configuration variables shows that you have set `TET_OUTPUT_CAPTURE=true` in the execute mode configuration. This supplies a default value of `TET_API_COMPLIANT=false`.
2. You will see that the TCM Start and the IC and TP Start and End lines are all marked ‘‘auto-generated by TCC’’. This shows that these lines were generated by `tcc` and not by the test case.

When you set `TET_API_COMPLIANT=false` (whether explicitly or by default), you are telling `tcc` that the test case or tool does not use the TETware API. So `tcc` doesn't copy output generated by API functions to the journal.

Normally you would set `TET_OUTPUT_CAPTURE=false` in the execute mode configuration when executing API-conforming test cases. If there is some reason why you want output capture mode enabled when executing API-conforming test cases, you should make the following assignments in the execute mode configuration:

```
TET_OUTPUT_CAPTURE=true
TET_API_COMPLIANT=true
```

See also

- ‘‘Test case structure’’ and ‘‘Configuration variables which modify TETware’s operation’’ in the TETware Programmers Guide.
- ‘‘TETware journal lines’’ in the TETware User Guide.

9. How to solve test case locking problems

Question

Occasionally when running `tcc` we get the following type of messages:

```
110|59 /tset/tc1/tc1 12:25:52|Build Start, scenario ref 24-0
50||(lock.c, 120): can’t acquire exclusive lock \
        c:/Tet3.2/suite/tset/tc1/tet_lock on system 000, \
        server reply code = ER_NOENT
130|59 -3 12:25:52|Build End, scenario ref 24-0
10|60 /tset/tc1/tc1 12:25:52|TC Start, scenario ref 24-0
50||(lock.c, 120): can’t acquire exclusive lock \
        c:/Tet3.2/suite/tset/tc1/tet_lock on system 000, \
        server reply code = ER_NOENT
80|60 -3 12:25:53|TC End, scenario ref 24-0
```

The lock appears to be a file in the test directory. Any ideas as to what might be causing this?

Answer

When `tcc` processes a test case, it uses a locking scheme to prevent multiple `tcc` processing threads from interfering with each other. The error messages that you describe are generated when `tcc` can’t create a lock for some reason.

Usually this is because either:

- the directory in which the lock is to be created doesn’t exist or is not writable; or:
- the lock already exists when `tcc` attempts to create it.

Clearly, the first case is evidence of a test suite setup problem. The directory containing a test case that is mentioned in a scenario must exist and be writable by `tcc` (in TETware-Lite) or `tccd` (in Distributed TETware).

Here are some common reasons for the second case:

a. The test suite is being processed by more than one instance of `tcc`.

b. The scenario uses the `:parallel:` directive and the test suite has not been structured in a way that permits parallel processing.

c. A previous `tcc` run has crashed or has been killed, leaving locks in place.

The solutions to these problems are:

a. Make sure that the test suite is only being processed by one instance of `tcc`. If you are using Distributed TETware, remember to check that your system is not being used as a remote target of `tcc` running on another system.

b. If you are using the `:parallel:` directive, you must arrange for each test case to have its own directory. If you use `:parallel,`*count*`:` to execute more than one copy of each test case, you must set `TET_EXEC_IN_PLACE=false` in the execute mode configuration.

c. Make sure that there is no other instance of `tcc` processing the test case, then remove the lock by hand.

See also

- ‘‘Build mode processing’’, ‘‘Execute mode processing’’, ‘‘Clean mode processing’’ and ‘‘Locking’’ in Chapter 3 of the TETware Programmers Guide.

10. Repeated scenario execution without modifying the scenario file

Question

A requirement of our manufacturing people is for `tcc` to be able to invoke repeated executions (for example: for hardware stress testing) without having to change the scenario files. Developers write the scenarios and don't want repetitive execution. But manufacturing, who wish to reuse the tests, do.

Answer

There are a couple of ways that you can do this using existing TETware functionality.

Method 1

(When you know how many times the manufacturing people want to repeat when you write the scenario.)

You can provide more than one scenario when you write the scenario file. One scenario can list all the tests, and the other can repeat the first scenario the required number of times.

For example:

```
manufacturer
        :repeat,100:^developer

developer
        /ts/tc1/tc1
        /ts/tc2/tc2
        ...
        etc.
```

Your developers can use the scenario called `developer` in the example above and the manufacturing people can use the scenario called `manufacturer`.

You can choose which scenario to execute on the `tcc` command-line. For example, to execute all the tests in the list once:

`tcc -ep` *test-suite-name* `developer`

or, to execute all the tests in the list 100 times:

`tcc -ep` *test-suite-name* `manufacturer`

Notes:

1. If one of the scenarios in the file is called `all`, you don't need to use the ''*test-suite-name scenario-name*'' style of syntax on the `tcc` command-line.
2. You can use `:timed_loop:` instead of `:repeat:` if your manufacturing people would find this more helpful. But don't do this until you are satisfied that the test cases are working reliably.

Method 2

(When you want to specify the number of times to repeat the scenario on the `tcc` command-line.)

In this method, you specify the basic scenario in a file and add a `:repeat:` directive on the command-line.

The scenario file should contain the non-repeating list of tests as in Method 1; for example:

```
developer
        /ts/tc1/tc1
        /ts/tc2/tc2
        . . .
        etc.
```

To execute all the tests once, you invoke `tcc` in the same way as in Method 1; for example:

```
tcc -ep test-suite-name developer
```

You can specify both the `−l` and `−s` options to process a scenario defined in a file under the control of a directive specified on the command-line. So, to repeat all the tests 50 times, you would say:

```
tcc -ep -s tet_scen -l ":repeat,50:^developer"
```

or, to repeat all the tests for (at least) 10 hours, you would say:

```
tcc -ep -s tet_scen -l ":timed_loop,36000:^developer"
```

Notes:

1. When you specify scenario lines using one or more `−l` options, `tcc` processes the lines as if they were in a scenario called `all`. So when you use `−l` and `−s` together, you can't have a scenario called `all` in the file specified by the `−s` option.

As can be seen from these examples, it is possible to invoke `tcc` with lots of different combinations of command-line options. It is usual to supply a shell script which contains the correct `tcc` invocation in cases where the command-line becomes too complicated to type in directly.

See also

- ''The scenario file'' in the TETware Programmers Guide.
- The `tcc` manual page in the TETware Users Guide.

11. Error message: `can't log on to TCCD on system` *n*

Question

When I try to run the distributed demo, `tcc` prints the message:

```
tcc (dtcc.c, 337): server connection closed (sysid = 0, pid = -1: STCC)
tcc (dtcc.c, 229): can't log on to TCCD on system 0
```

I am running Distributed TETware on a UNIX system and using the **inetd** version of `tccd`.

Answer

The first message indicates that the connection from `tcc` was accepted by `inetd` and then closed for some reason.

Possibilities are:

1. If `inetd` could not execute `in.tccd` for some reason when `tcc` connected to the well-known **tccd** port, you should see an error message from `inetd` in the `syslog` file.
2. If `in.tccd` started up but then exited with an error, you should see a startup message followed by an error message in the `tccd` log file (usually `/tmp/tccdlog`).

Additional information

The ‘‘can't log on to TCCD’’ message may be preceded by other messages. One example is the ‘‘tcc/tcp: unknown service’’ message that is described in another Knowledge Base article.

Another example is as follows:

```
tcc (logon.c, 133): server error (sysid = -1, pid = 12961: STCC)
tcc (dtcc.c, 229): can't log on to TCCD on system 1
```

In this case it is necessary to check the `/tmp/tccdlog` file on system 1 for further information about the error. It contained the lines:

```
tccd (12961) 28 May 14:09:25: connection received from texel
tccd (12961) 28 May 14:09:25 (tccd_in.c, 418): can't open \
        /home/tet/systems.equiv: No such file or directory
tccd (12961) 28 May 14:09:25 (tccd.c, 398): client connection closed \
        (sysid = 0, pid = 2234: MTCC)
```

This shows that the reason for the failure is because the `systems.equiv` file has not been set up correctly on system 1.

See also

- ‘‘Starting `tccd`’’ in the TETware Installation Guide for UNIX Operating Systems.
- ‘‘The Distributed C API demonstration’’ in the TETware User Guide.
- The `tccd` manual page in the TETware User Guide.

12. How to run test cases in a known environment

You can use an exec tool to do this. For example, suppose you want to make sure that a test case always executes in the C locale. The exec tool might look like this:

```
#!/bin/sh

# set and export the variables
LANG=C
LC_CTYPE=C
LC_MESSAGES=C
LC_NUMERIC=C
LC_TIME=C
export LANG LC_CTYPE LC_MESSAGES LC_NUMERIC LC_TIME

# then execute the test case
exec "$@"
```

Then, put the following assignment in the execute mode configuration file:

`TET_EXEC_TOOL=`*exec-tool*

where *exec-tool* is the name of the shell script that you have just created.

When you specify the name of an exec tool, it should be either:

— relative to the test case execution directory; or:

— an absolute path name; or:

— in one of the directories listed in the `PATH` environment variable.

Points to note:

a. A name relative to the test case execution directory won't work unless you have `TET_EXEC_IN_PLACE=true` in the execute mode configuration.

b. On a Win32 system the name of a shell script should have a `.ksh` suffix. Or you can say:

   `TET_EXEC_TOOL=sh`
   `TET_EXEC_FILE=`*exec-tool*

   in the execute mode configuration.

c. On a Win32 system a shell script should not start with a `#!` line.

There follows an example of a more sophisticated exec tool:

Question

We have a problem where different installations by different users often have a different environment which can affect the test results obtained. We would like to be able to specify a set of environment variables in a file and have `tcc` put them in the environment when test cases are executed.

Answer

You can use an exec tool to do this. In the following example, the exec tool gets the name of the environment file to use from a variable called `TS_ENV_FILE` in the execute mode configuration.

```
#!/bin/sh
#
# exec tool which executes a test case with environment taken from
# the file defined by the TS_ENV_FILE configuration variable
#
# extract the value of TS_ENV_FILE from the configuration for the
# current mode of operation -
# ignore blank lines and comments in the config file
#
# (the value of TET_CONFIG is set by tcc before invoking the build tool)
TS_ENV_FILE=
eval `sed -n 's/#.*//
        /^[     ]*\$/d
        /^TS_ENV_FILE=/s/\([^=]*\)=\(.*\)/\1="\2"/p' ${TET_CONFIG:?}`

# if a TS_ENV_FILE has been defined, read it in
if test ! -z "$TS_ENV_FILE"
then
        if test -r $TS_ENV_FILE
        then
                set -a
                . $TS_ENV_FILE
        else
                echo "$0: can't read environment file $TS_ENV_FILE" 1>&2
                exit 1
        fi
fi

# finally, execute the test case
exec "$@"
```

Then put the following lines in the execute mode configuration file:

`TET_EXEC_TOOL=`*exec-tool*
`TS_ENV_FILE=`*env-file*

When you do this, environment variables defined in *env-file* will be put in the environment when test cases are executed.

See also

- ‘‘Execute mode processing’’ and ‘‘Configuration variables which modify TETware’s operation’’ in the TETware Programmers Guide.
- Users of Win32 systems should check out the section entitled ‘‘Executable files’’ in Appendix H of the TETware User Guide.

13. Problems when running the perl demo

Question

I am running Distributed TETware on a UNIX system.

When I run the perl demo, no test case output appears in the journal file.

For example:

```
70||"starting scenario"
110|0 /ts/tc1 10:59:49|Build Start, scenario ref 2-0
130|0 0 10:59:50|Build End, scenario ref 2-0
10|1 /ts/tc1 10:59:50|TC Start, scenario ref 2-0
80|1 255 10:59:54|TC End, scenario ref 2-0
300|2 /ts/tc1 10:59:54|Clean Start, scenario ref 2-0
320|2 0 10:59:56|Clean End, scenario ref 2-0
110|3 /ts/tc2 10:59:56|Build Start, scenario ref 3-0
130|3 0 10:59:57|Build End, scenario ref 3-0
10|4 /ts/tc2 10:59:57|TC Start, scenario ref 3-0
80|4 255 11:00:00|TC End, scenario ref 3-0
300|5 /ts/tc2 11:00:01|Clean Start, scenario ref 3-0
320|5 0 11:00:02|Clean End, scenario ref 3-0
70||"next is the last test case"
110|6 /ts/tc3 11:00:02|Build Start, scenario ref 5-0
130|6 0 11:00:03|Build End, scenario ref 5-0
10|7 /ts/tc3 11:00:03|TC Start, scenario ref 5-0
80|7 255 11:00:07|TC End, scenario ref 5-0
300|8 /ts/tc3 11:00:07|Clean Start, scenario ref 5-0
320|8 0 11:00:08|Clean End, scenario ref 5-0
70||"done"
900|11:00:08|TCC End
```

The following errors appear in the `/tmp/tccdlog` file:

```
tccd (25135) 29 Sep 10:59:51 (tcfexec.c, 205): \
        can't exec /export/home0/tet/perldemo/tet_tmp_dir/25131a/tc1: \
        No such file or directory
tccd (25143) 29 Sep 10:59:57 (tcfexec.c, 205): \
        can't exec /export/home0/tet/perldemo/tet_tmp_dir/25131a/tc2: \
        No such file or directory
tccd (25154) 29 Sep 11:00:04 (tcfexec.c, 205): \
        can't exec /export/home0/tet/perldemo/tet_tmp_dir/25131a/tc3: \
        No such file or directory
```

There is nothing in `tet_tmp_dir`. Is there something that I need to add in the setup?

Answer

The perl demo does not have a setting for `TET_EXEC_IN_PLACE` in the execute mode configuration file `tetexec.cfg`. When `TET_EXEC_IN_PLACE` is undefined, its default value is `False`.

When `TET_EXEC_IN_PLACE` is false, `tcc` copies the contents of the test case directory to a temporary directory below `tet_tmp_dir` and executes the test case from there. The temporary directory is removed when execution finishes. This is why you won't see anything below `tet_tmp_dir` after `tcc` exits.

You will notice that, in the journal, the exit status in each Test Case End line is 255. This shows that the test case could not be executed and corresponds to the ‘‘can’t exec’’ messages in the `/tmp/tccdlog` file. Here are some possible reasons why the test cases could not be executed:

1. **Problem**
   The test cases are missing from the test case source directory, so don’t get copied to the temporary directory before execution. Thus the exec fails.

   Solution
   Check that the test cases `tc1`, `tc2` and `tc3` exist in `$TET_ROOT/contrib/demo/ts`. If they are missing, install them from the `contrib` distribution.

2. **Problem**
   Each test case in the perl demo must be interpreted by `perl`. In the demo this is achieved by the line `#!/usr/bin/perl` at the top of each test case file. If `perl` is not installed in `/usr/bin` on your system, the symptoms will be the same as if the test cases are missing.

   Solution
   Check that the file `/usr/bin/perl` exists on your system (and is executable). If it doesn’t, you can do **one** of the following (the most recommended is first).
   Either:

   a. Install a symlink named `/usr/bin/perl` which points to the location of the perl executable on your system; or:

   b. Set `TET_EXEC_TOOL` in `tetexec.cfg` to the location of `perl` on your system. For example, if `perl` lives in `/usr/local/bin` on your system, you would say:

      ```
      TET_EXEC_TOOL=/usr/local/bin/perl
      ```

      or:

   c. Change the `#!` line in every test case to refer to the location of `perl` on your system.

Needless to say, you must have `perl` installed somewhere on your system in order to have any chance of running the perl demo!

Win32 systems

A Win32 system does not interpret `#!` in a script file. Instead, the o/s uses the file name suffix indicate how a file should be executed. On a Win32 system the TETware execution subsystem understands that a file with a `.pl` suffix should be interpreted by `perl`. In order for this to work it is necessary for the directory containing `perl.exe` to be included in the value of the `PATH` environment variable.

See also

- ‘‘Execute mode processing’’ and ‘‘Configuration variables which modify TETware’s operation’’ in the TETware Programmers Guide.
- Users of Win32 systems should check out the section entitled ‘‘Executable files’’ in Appendix H of the TETware User Guide.

14. Executing a set-UID test case

This information is not applicable to Win32 systems.

Question

When I set the set-UID bit on a test case, it doesn't change the effective user ID when the test case is executed by `tcc`.

Answer

If you specify `TET_EXEC_IN_PLACE=false` in the execute mode configuration, `tcc` copies the test case to a temporary directory and executes it from there. The act of copying the test case changes its owner and clears the set-UID bit as well.

If you don't specify `TET_EXEC_IN_PLACE` its value defaults to false, so the effect is the same.

So, if you want `tcc` to execute a set-UID test case, you must specify `TET_EXEC_IN_PLACE=true` in the execute mode configuration file.

If there is some reason why you don't want to execute test cases from the source directory, you can specify an **alternate execution directory** and have `tcc` execute them from there.

See also

- ''Directory structure'', ''Execute mode processing'' and ''Configuration variables which modify TETware's operation'' in the TETware Programmers Guide.
- ''Environment variables'' in the TETware User Guide.

15. How to get the standard error from an API-conforming test case to appear in the journal

Question

I have a number of tests where the test program writes to `stderr`. I would like this information to appear in the journal file. I have been unable to achieve this using the `TET_OUTPUT_CAPTURE` settings. So I have resorted to redirecting `stderr` to a file and running a small program which reads in each line of the file and calls the function `tet_infoline()`.

Is there a better way to achieve this?

Answer

It is possible to instruct `tcc` to capture `stdout` and `stderr` from a test case and copy it to the journal. When using this functionality, it is helpful to understand the interaction between the `TET_OUTPUT_CAPTURE` and `TET_API_COMPLIANT` configuration variables.

You can run your test case with `TET_OUTPUT_CAPTURE=true` in the execute mode configuration. When you do this, `tcc` will execute test cases with output capture mode enabled. However, setting `TET_OUTPUT_CAPTURE=true` also has the effect of providing a default value of `TET_API_COMPLIANT=false`. So if your test case uses the API, you will need to set `TET_API_COMPLIANT=true` explicitly, otherwise you won't get any information lines or result lines in the journal.

See also

- ''Execute mode processing'' and ''Configuration variables which modify TETware's operation'' in the TETware Programmers Guide.

16. How to run a distributed test case which reboots one of the systems

The information in this article is not presented as a complete solution but might be helpful to someone who is attempting to solve a similar type of problem.

Question

We are using TETware 3.2 for running distributed tests on UNIX systems using the `:remote:` directive. For example: `:remote,000,001,002:` where 000 is the master and 001, 002 are two other systems participating in the distributed test.

We have a requirement where we need to shutdown one of the systems that is running the test.

1. Will `tcc` on the master system hang or report `ER_TIMEDOUT` or any such messages because one of the systems is shutdown?

   Can the other systems and master continue to run the test?

2. Is it possible for the system to re-join the test if it is rebooted again?

3. Assuming I don't include the sysid in a call to `tet_remsync()` after the system is shutdown, will there be problems with the automatic sync calls that are performed by the API?

Answer

First some background . . .
`tcc` maintains a connection with `tccd` on each system for the lifetime of the scenario. The test case on each system has a connection to `tetsyncd` and `tetxresd` on the master system.

The precise behaviour that you will observe depends on what TCP/IP does when the machine at the other end shuts down. If the machine that is shutting down closes the connections in an orderly way (as would happen in a normal shutdown), then the connected peers will get notification of the close in the normal way (EOF on read, `SIGPIPE` on write). Each process (`tcc`, `tetsyncd`, `tetxresd`) that sees a connection close will regard this as an error condition and will take appropriate action. In the case of `tetsyncd`, subsequent attempts by the other test case parts to perform sync operations (automatic or user-defined) will fail because when the connection closes, `tetsyncd` marks the system's sync state as DEAD.

By contrast, if the connections are not closed in an orderly way (as can sometimes happen when a machine crashes), the connection will simply hang for some period of time. Synchronisation requests will time out, but other connections will wait indefinitely for something to happen to the connection.

Now, to answer your questions . . .

1. The other systems will not be able to continue to run the test. Test cases on the other systems will fail with an error condition at the next automatic sync point.

2. It is not possible for the system to re-join the test after it has rebooted. Since TCP is used for the inter-process connections (which is stateful), there is no way to restore the connection after a reboot.

3. The automatic sync calls will fail after one of the systems is rebooted. There is no mechanism for deleting a participating system from an autosync event part-way through a test case's execution.

So, if you want to reboot (say) system 2, you should not include system 2 in the system list that you pass to the `:remote:` directive.

Perhaps you could try the following:

a. Instead, you can call `tet_remexec()` from a child process on system 1. When you do this, the API in the child process will set up its own connection to system 2. This will prevent the API in your test case from retaining state information about system 2. Be sure to do nothing in the parent process which would cause the API to connect to system 2 before you call `tet_remexec()` from the child. (Basically this means not calling `tet_remexec()` or `tet_remtime()` with a sysid argument of 2 from the parent.

b. To create a child process, simply call `tet_fork()` with a NULL `parentproc` argument and then call `tet_remexec(2, ...)` from the `childproc` function. (You should specify a zero `validresults` argument and a suitably short timeout — say 30 seconds.)

c. By the time that `tet_remexec()` returns, the remote process will have started. So you can then immediately call `tet_exit()` from the child process on system 1. (The child process should exit with zero status if `tet_remexec()` succeeded and non-zero if `tet_remexec()` failed.) This will log off all the connected servers (in particular: the `tccd` on system 2) and exit. At this point the call to `tet_fork()` will return in the parent on system 1. The return value of `tet_fork()` will indicate whether or not the call to `tet_remexec()` was successful in the child.

d. Now you must make sure that the remote process on system 2 waits for the child process on system 1 to exit. Note that when the child process on system 1 exits, it will log off `tccd` on system 2 first. When `tccd` sees the logoff it will send a `SIGHUP` signal to the unwaited-for process that was started by `tet_remexec()`. So you should be sure to ignore `SIGHUP` in this process. You will need to wait until the child process on system 1 exits (thus closing the connection to `tccd`). Then call `tet_logoff()` to close the connections back to the `tetsyncd` and `tetxresd` servers on the master system. Finally you can call `reboot()` to reboot system 2.

You will need call `tet_remsync()` at various times so as to ensure that all this happens in the correct order.

The order of events will look something like this. Events that are synchronised are connected by ⟷.

System 1		**System 2**
Create a child process using `tet_fork()` with a NULL `parentproc` and zero `validresults` (parent blocks in `tet_fork()` call, waiting for child to exit)		
<u>In child process</u>		
Call `tet_remexec()` to launch a remote process on system 2 that will reboot the system	⟷	`tccd` forks and execs the remote process
`tet_remexec()` returns (if `tet_remexec()` returns –1, don't sync but print a diagnostic and call `tet_exit(1)`)	⟷	Remote process controller calls `tet_main()`
		<u>In remote process</u>
		Call `signal(SIGHUP, SIG_IGN)`
Sync with system 2 to syncpoint N (sync call returns)	⟷	Sync with system 1 to syncpoint N+1 (sync call blocks)
Call `tet_exit(0)` (child logs off `tccd` on system 2 and exits)	⟷	(`tccd` sends `SIGHUP` to remote process which is ignored — process stays blocked in sync call)
(call to `tet_fork()` returns in parent — if child exit status is non-zero this means that `tet_remexec()` has failed so give up; the API has already reported `UNRESOLVED` in this case)		
<u>Parent process continues</u>		
Sync with system 2 to syncpoint N+1	⟷	(sync call returns)
Sleep a bit — wait for system 2 to call `reboot()`		Call `tet_logoff()` (no more API calls are allowed after this point!)
		Call `reboot()` (remote process and `tccd` get killed as system 2 goes down)
Enter the `ping`/`sleep` loop — wait for remote system to come back up again		

System 1		System 2
`ping` loop ends	⟷	System restarts — a new instance of `tccd` becomes available once the system comes up multi-user
Sleep a bit — wait for system 2 to come up multi-user		
Call `tet_remexec()` to launch a different remote process on system 2; this time, call `tet_remwait()` to wait for the remote process to terminate		etc . . .

Footnote

This suggestion was offered speculatively and had not been tried out at the time of writing. But a subsequent message from the recipient indicated that a strategy based on this suggestion had in fact been successful.

See also

- The descriptions of `tet_fork()`, `tet_remsync()`, `tet_remexec()`, `tet_exit()` and `tet_logoff()` in Chapter 8 of the TETware Programmers Guide.
- ‘‘Distributed TETware architecture’’ and ‘‘TETware programs’’ in the TETware User Guide.

17. How to link the `tet_main()` function when using the C++ API

Question

When I build a child process that contains a `tet_main()` function, I get the following error messages when from the link stage. I am using MSVC++ version 5.0 on Windows NT.

```
Linking...
libapi.lib(child.obj) : error LNK2001: unresolved external
symbol _tet_main
Debug/x.exe : fatal error LNK1120: 1 unresolved externals
Error executing link.exe.

x.exe - 2 error(s), 18 warning(s)
```

My program heading looks like this:

```
int tet_main(int argc, char **argv)
{
        .....
}
```

Answer

When you use the C++ API, your `tet_main()` function should have C linkage in order to enable the linker to resolve the symbol correctly. In a C++ program you give a function C linkage by putting it inside an `extern "C"` code block.

For example:

```
extern "C" {

        int tet_main(int argc, char ** argv)
        {
                // whatever you want here
        }

}
```

See also

- ‘‘The C++ API’’ in the TETware Programmers Guide.

18. Problems with `SIGCHLD` in the Perl API

This information is not applicable to Win32 systems.

Question

When executing a perl API test, I'm running into problems with the TCM and `SIGCHLD` when I invoke other processes via the perl `system()` call. After the `system()` function completes, the test aborts due to receipt of the `SIGCHLD` signal.

I tried adding `@tet'sig_ignore(18)` to my code, but apparently `SIGCHLD` is considered an uncatchable signal by `tcm.pl`.

Answer

This is correct — if a TCM catches `SIGCHLD` it can't reap child processes correctly (as you have observed).

The setting up of the signal lists in the perl API is performed by a configuration script when the API is installed. By default, the TCM catches all the signals not in the **special signal** list. One reason for the behaviour that you have observed might be that some problem during installation has prevented the list from being set up correctly. Or you might be using a copy of the perl API that has been configured for use on another machine where `SIGCHLD` has a different numerical value.

You should try reinstalling the perl API from the source and configuring it on your machine.

See also

- ''Building TETware'' in the TETware Installation Guide for UNIX Operating Systems.

19. How to handle POSIX signals in a C language test case

This information is not applicable to Win32 systems.

Question

I'm using the C API to port some legacy test code to TET. The code issues a signal 14 (`SIGALRM`) during normal execution and this is trapped by the TCM, terminating the test prematurely.

I've tried setting `TET_SIG_IGN=14` and `TET_SIG_LEAVE=14` in the execute mode configuration but the TCM tells me this is an illegal entry. Are there any workarounds?

Answer

`SIGALRM` is a **standard** signal; that is: a signal whose behaviour is defined by POSIX. The TCM does not permit the handling of standard signals to be altered by the configuration variables that you mention because the way that such signals are to be handled is considered to be a matter for the test suite author rather than for the user.

If you want to change the TCM's default signal handling in a particular test purpose function, you can add code to change the signal's disposition in the function itself. This is the preferred solution — it enables each test purpose function to be self-contained and not rely on the execution (or non-execution) of a previous test purpose function.

Alternatively you can change the signal's disposition in the test case startup function, then set the API's global variable `tet_nosigreset` to a non-zero value in the startup function. When you do this, the TCM does not reset the dispositions of signals before it calls each test purpose function and so these dispositions remain unchanged (by the TCM, at least) throughout the life of the test case.

See also

- The descriptions of `tet_nosigreset`, `TET_SIG_IGNORE` and `TET_SIG_LEAVE` in Chapter 8 of the TETware Programmers Guide.
- ‘‘The Test Case Manager’’ in the TETware Programmers Guide.

20. How to create multiple processes using `tet_fork()`

This information is not applicable to Win32 systems.

Question

I have a need to fork multiple process. After forking multiple processes, the parent should wait for all these processes.

Typically my code should like this:

```
for (i = 0; i < 10; i++) {
        if ((pid = fork()) == 0)
                child_func();
}

for (i = 0; i < 10; i++) {
        waitpid(.......);

        /* check for the child’s return status... */
}

...
```

If I use `fork()` I am not able to use API functions in the child processes. If I use `tet_fork()` I cannot create more than one child process at once. Is there any way to do this?

Answer

You can do this using recursive calls to a `(*parentproc)()` function.

For example:

```
#include <stdlib.h>
#include <tet_api.h>

void (*tet_startup)() = TET_NULLFP, (*tet_cleanup)() = TET_NULLFP;
static void tp1(), tp1_child(), tp1_parent();

struct tet_testlist tet_testlist[] = {
        { tp1, 1 },
        { TET_NULLFP, 0 }
};

static int tp1_fcount;
static int testfail;

static void tp1()
{
        testfail = 0;

        tp1_fcount = 0;
        tp1_parent();

        if (!testfail)
```

```
                tet_result(TET_PASS);
}

static void tp1_parent()
{
        int level;
        void (*parentproc)();

        if ((level = ++tp1_fcount) < 10)
                parentproc = tp1_parent;
        else
                parentproc = TET_NULLFP;

        (void) tet_printf("about to call tet_fork(): level = %d", level);
        if (tet_fork(tp1_child, parentproc, 30, 0) < 0) {
                /* API prints an infoline and generates a result */
                testfail++;
        }
        else
                (void) tet_printf("tet_fork() succeeded, level = %d", level);
}

static void tp1_child()
{
        (void) tet_printf("in child process, PID = %d", getpid());
        tet_exit(0);
}
```

Question

How to I extend this example to an arbitrary number of child processes without creating an infinite number of stack frames?

Answer

This is a rather different situation.

You can call `tet_fork()` with a waittime of −1. In this case, the API does not wait for the child process and the `validresults` argument is ignored.

When this feature is used, the `parentproc` function is supposed to wait for the child. If the child is still running when the `parentproc` function returns, the API kills the child process. So you have to fool the API by providing a dummy child for it to kill; that way your child process is still running when `tet_fork()` returns. The dummy child process must not call any API functions.

Then it is your responsibility to make sure that you have waited for all the child processes to exit before the test purpose returns control to the TCM.

For example:

```
#include <stdlib.h>
#include <errno.h>
#include <signal.h>
#include <tet_api.h>

void (*tet_startup)() = TET_NULLFP, (*tet_cleanup)() = TET_NULLFP;
static void tp1(), tp1_child(), tp1_parent();

struct tet_testlist tet_testlist[] = {
        { tp1, 1 },
        { TET_NULLFP, 0 }
};

static int testfail;
static pid_t child_pid;

void tp1()
{
        testfail = 0;

        for (;;) {
                /*
                ** you will need some code here to wait for some event
                ** and/or break out of the loop
                */
                tet_infoline("parent: about to call tet_fork()");
                if (tet_fork(tp1_child, tp1_parent, -1, 0) < 0) {
                        /* API prints a diagnostic */
                        testfail++;
                        break;
                }
                (void) tet_printf("parent: child PID = %d", child_pid);
        }

        /*
        ** you must ensure that all the child processes have exited
        ** at this point
        */

        if (!testfail)
                tet_result(TET_PASS);
}

static void tp1_parent()
{
        int pid;

        /* create a dummy child for the API to kill */
        switch (pid = fork()) {
        case 0:
                /* child must not call any API functions */
                (void) signal(SIGTERM, SIG_DFL);
```

```
                pause();
                _exit(0);
                break;
        case -1:
                /* real trouble here - the only safe thing to do is to
                        exit from the test case */
                (void) tet_printf("fork() failed, errno = %d", errno);
                tet_exit(1);
        }

        /* arrange for the API to kill the dummy child instead of the
                childproc function */
        child_pid = tet_child;
        tet_child = pid;
        return;
}

static void tp1_child()
{
        (void) tet_printf("in child process, PID = %d", getpid());

        /* whatever you want here */

        tet_exit(0);
}
```

Not that this example does not contain any code to wait for the child processes, or to cause the main loop to end. So if you compile and run the code as it stands, you will run out of processes at some point.

See also

- ‘‘Generating and executing processes’’ in Chapter 8 of the TETware Programmers Guide.

21. Porting Korn Shell arithmetic expressions from UNIX to Win32 systems

Question

I have ported a Korn Shell test case from a UNIX system to a Win32 system. When I run the test case it prints the following message:

```
i+=4: tp4: d:/TET/tet32/lib/ksh/tcm.ksh 678: .: create.ksh 83: not found
```

The file is indeed present:

```
-rwxrwxrwa   1 Administrators  ENG-NT\staff \
        18341 Jul 30  1997 d:/TET/tet32/lib/ksh/tcm.ksh
```

Answer

I don't think that the problem is to do with the shell not being able to find `tcm.ksh`.

I think that you have to read the diagnostic backwards like this:

`create.ksh` line 83 calls `tcm.ksh`
`tcm.ksh` line 678 calls function `tp4`
`tp4` calls `i+=4`
shell reports `command not found`

This suggests that the shell is interpreting the line `i+=4` as a command whereas I expect that you intended it to be an arithmetic expression. This is because the MKS Shell is a POSIX shell and doesn't support the Korn Shell extensions by default. In the Korn Shell, you can say:

```
(( i+=4 ))
```

which is the same as saying:

```
let "i+=4"
```

But in the POSIX shell, `(( i+=4 ))` simply runs the command `i+=4` in a subshell.

One way to write an arithmetic expression that is portable between the Korn Shell on UNIX systems and the MKS Shell on Win32 systems is:

```
: $(( i += 4 ))
```

Or you can say:

```
case ‘uname -s‘ in
Windows_NT|Windows_95)
        set -K
        ;;
esac
```

to get Korn Shell behaviour from the MKS Shell on Win32 systems.

22. Running `tccd` from the command line

This information is not applicable to Win32 systems.

Most modern UNIX systems support `inetd` so most people will build the **inetd** version of `tccd` (the Test Case Controller daemon). However, there are some situations where it is required to run `tccd` from the command line. If you need to do this for some reason it is best to build the **rc** version of `tccd`. Some hints about how best to do this are presented in this article.

When you run `tccd` from the command-line, it inherits all your environment variables. This can result in test cases being influenced by your environment in a way that can't be repeated on someone else's system. This can cause a lot of trouble if you develop a test suite that works OK for you, but fails in various ways when you ship it to a customer. In order to ensure that you don't fall in to this trap when developing a test suite, it is necessary to start `tccd` with a known clean environment.

You can use the `env` command to do this, and put the correct invocation in a shell script.

For example:

```
#!/bin/sh
exec env - PATH=$PATH TZ=$TZ ... tccd [options ...]
```

The – argument to the `env` command cleans out the environment for the command to be executed. Then you should specify just the list of environment variables that you actually need to run test cases.

This issue is less of a problem when starting `tccd` on the local system (system 0) because `tcc` sends all of its environment to `tccd` on system 0 when it logs on. But it is important to run `tccd` in a known environment on remote systems, because in this case the environment is not sent. In the past a number of people have been caught out by not taking care of this issue when starting `tccd` from the command-line.

If you run `tccd` as yourself, it will be unable to change its user ID to `tet`. This results in an error message in the `/tmp/tccdlog` file. However, `tccd` will still run (using your user ID) provided your user ID and group ID are each >= 100.

You can use the command:

```
tccd -u your-login-name
```

if you want to avoid the error message being printed.

See also

- ''Starting `tccd`'' in the TETware Installation Guide for UNIX Operating Systems.
- The `tccd` manual page in the TETware User Guide.

23. How to determine the value of an XTI address string

This information is not applicable to Win32 systems.

Most modern UNIX systems provide support for the socket network interface. Most people will build Distributed TETware to use the socket network interface and so don't need to be concerned about XTI addresses. The information presented in this article might be useful if you have to set up Distributed TETware to use the XTI network interface for some reason.

Where does TETware use an XTI address string?

When Distributed TETware is built to use the XTI network interface you have to specify an XTI address string in the following places:

1. `tccd` must be invoked with a `−p` option which tells it on which network end point to listen.
2. For each entry in the `systems` file you must provide a third field that specifies the XTI address that can be used to connect to `tccd` on that system.

XTI address strings

An XTI address string consists of a sequence of 2-digit hexadecimal values. When TCP is the transport provider, these values often represent a dump of a `sockaddr_in` structure which describes the network address.

For example, consider the following definitions taken from `<netinet/in.h>` on a hypothetical machine:

```
struct in_addr {
        unsigned long s_addr;
};

struct sockaddr_in {
        short sin_family;
        unsigned short sin_port;
        struct in_addr sin_addr;
        char sin_zero[8];
};
```

Suppose that on this machine:

— a `short` is 16 bits wide

— a `long` is 32 bits wide

— structure members are aligned on 2-byte boundaries (so there will be no padding between members in the `sockaddr_in` structure)

It can be seen that the format of an XTI address string on this machine will be:

FFFFPPPPAAAAAAAA`0000000000000000`

where *FFFF* is the address family (in host byte order) *PPPP* is the port number (in network byte order) and *AAAAAAAA* is the IP address (also in network byte order).

`tccd -p` option

In most cases you want `tccd` to accept connections on all network interfaces, so you need to specify the IP address as `INADDR_ANY` (value zero on many systems).

Suppose that:

— the byte order of the machine is high byte first (big-endian)

— you decide to use port 7500 as the well-known port

— the name of the transport provider is `/dev/tcp`

You would invoke `tccd` as follows:

`tccd -p 00021d4c000000000000000000000000 -M TCP -P /dev/tcp` [*other-options* ...]

This XTI address string can be read as follows:

`0002` The address family (`AF_INET` — value 2 on many systems)

`1d4c` The port number to listen on (7500 decimal is 0x1d4c hex)

`00000000`
The IP address to use (`INADDR_ANY` — value zero on many systems)

`0000000000000000`
Zero-filled padding (`char sin_zero[8]`)

Note that the IP address and port number are in network byte order. So on a little-endian machine the only thing that changes is the address family. For example:

`tccd -p 02001d4c000000000000000000000000 -M TCP -P /dev/tcp` [*other-options* ...]

`systems` file

The XTI address in a systems file entry is constructed in the same way as for `tccd` except it is necessary to specify a real IP address instead of `INADDR_ANY`. Foe example, suppose you decide to use a machine called `fred` as system 1. The IP address of `fred` is 89.0.173.24 and `tccd` is listening on port 7500. The systems file entry would look like this:

```
001     fred    00021d4c5900ad180000000000000000
```

This XTI address string can be read as follows:

`0002` The address family (`AF_INET` — value 2 on many systems)

`1d4c` The port number to listen on (7500 decimal is 0x1d4c hex)

`5900ad18`
The IP address to use (89.0.173.24 is 0x5900ad18 hex)

`0000000000000000`
Zero-filled padding (`char sin_zero[8]`)

See also

- ‘‘Systems definitions’’ in the TETware Programmers Guide.
- The `systems` and `tccd` manual pages in the TETware User Guide.

24. Variable types used in TETware

TETware uses the following types of variables:

1. Environment variables.
   These are variables that are read from the environment when `tcc` is invoked.

2. Configuration variables.
   These are variables read from configuration files that you must provide.

3. Communication variables.
   These are variables that `tcc` puts in the environment when it executes a test case or tool.

These types of variable are logically distinct. In particular, it should be understood that configuration variables and environment variables are not the same.

Configuration variables

These variables are used by `tcc` and may also be accessed by API-conforming test cases and tools. When the C API is used, configuration variables may be accessed by calling the `tet_getvar()` API function. When the Shell or Korn Shell API is used, the TCM makes configuration variables available as readonly shell variables but does not export them.

In Distributed TETware it is possible to define configuration variables with different values on different systems.

Distributed configuration variables

As with configuration variables, the Distributed `tcc` reads distributed configuration variables from a file that you must provide. However, the APIs do not make distributed configuration variables available to test cases and tools.

Communication variables

These are environment variables that `tcc` uses to pass information to TCMs. They can be accessed by test cases and tools but should not be modified since the operation of the APIs depend on them.

See also

- ‘‘Communication variables’’ and ‘‘Configuration files’’ in the TETware Programmers Guide.
- ‘‘Environment variables’’ in the TETware User Guide.
- The `tcc` manual page in the TETware User Guide.

25. Displaying scenario trees with `tetscpp`

There is a useful program that you can build which shows you how the `tcc` scenario parser interprets a scenario file. It is not part of the official TETware release (and therefore is not supported) but it enables you to look at the scenario tree without having to go to the trouble of getting `tcc` to execute the scenario and then digging around in journal and debug output.

The program is called `tetscpp` and can be built in the `tcc` source directory.

To build this program:

```
cd $TET_ROOT/src/tet3/tcc
make tetscpp
```

The syntax of `tetscpp` is as follows:

> `tetscpp` [`−P`] [`−c` *compat-mode*] [`−o` *output-file*] [`−r`] [`−s` *scenario*] [`−t` *tabwidth*] [`−y` *string*] [`−n` *string*] [*files* ...]

`tetscpp` reads scenarios from the named *files*. If no *files* are specified, `tetscpp` reads scenarios from the standard input.

The following options are understood:

`−P`	Do not emit `cpp`-style line control directives.
`−c` *compat-mode*	Select compatibility mode. *compat-mode* should be one of `d` (for dTET2 mode) or `e` (for ETET mode).
`−o` *output-file*	Leave the output in the specified *output-file* instead of printing it on the standard output.
`−r`	Include scenario reference numbers in the output.
`−s` *scenario*	Parse the specified *scenario* instead of the one named `all`.
`−t` *tabwidth*	Specifies the width of a tab when indenting output. Defaults to 4 spaces.
`−y` *string* `−n` *string*	Meanings are the same as for `tcc`.

In addition, trace options may be specified using `−T` in the usual way. When the `tet_Tscen` flag (flag indicator `p`) is non-zero, the output includes the address of the scentab element from which each output line is derived. (Trace options are described in the appendix entitled ‘‘Trace and debugging facilities’’ in the TETware User Guide.)

`tetscpp` does not have a concept of a **test suite root** directory, so its handling of scenario include files is rather primitive. If a scenario contains an include file name, `tetscpp` simply interprets the name relative to its current working directory.

Example 1

Here is the scenario from the distributed demonstration test suite that is described in the TETware Programmers Guide:

```
all
        "starting scenario"
        :remote,000,001:
        /ts/tc1
        /ts/tc2
        "next is the last test case"
        /ts/tc3
        :endremote:
        "done"
```

The following output is generated when this scenario is processed by `tetscpp -P`:

```
scenario("all")
{
    sceninfo("starting scenario");
    remote(0, 1)
    {
        testcase("/ts/tc1", "all");
        testcase("/ts/tc2", "all");
        sceninfo("next is the last test case");
        testcase("/ts/tc3", "all");
    }
    sceninfo("done");
}
```

Example 2

Consider the following two scenarios. At first inspection they appear to be different but analysis with `tetscpp` shows that `tcc` would process them in exactly the same way.

Scenario 1:

```
all
        :timed_loop,18000:
            :parallel:
                :repeat,10:
                    /tset/test1/tc1
                    /tset/test2/tc2
                    /tset/test3/tc3
                :endrepeat:
                :repeat,30:
                    :random:
                        /tset/test4/tc4
                        /tset/test5/tc5
                        /tset/test6/tc6
                        /tset/test7/tc7
                    :endrandom:
                :endrepeat:
            :endparallel:
        :endtimed_loop:
```

Scenario 2:

```
all
        :timed_loop,18000:^loop1
loop1
        :parallel:^list1
loop2
        :repeat,10:^list2
loop3
        :repeat,30;random:^list3
list1
        ^loop2
        ^loop3
list2
        /tset/test1/tc1
        /tset/test2/tc2
        /tset/test3/tc3
list3
        /tset/test4/tc4
        /tset/test5/tc5
        /tset/test6/tc6
        /tset/test7/tc7
```

The way that `tcc` would process either of these scenarios can be shown by executing the following command:

`tetscpp -Pce` *scenario-file*

The output from `tetscpp` is as follows:

```
scenario("all")
{
    timed_loop(18000)
    {
        parallel(1)
        {
            sequential()
            {
                repeat(10)
                {
                    testcase("/tset/test1/tc1", "all");
                    testcase("/tset/test2/tc2", "all");
                    testcase("/tset/test3/tc3", "all");
                }
            }
            sequential()
            {
                repeat(30)
                {
                    random()
                    {
                        testcase("/tset/test4/tc4", "all");
                        testcase("/tset/test5/tc5", "all");
                        testcase("/tset/test6/tc6", "all");
                        testcase("/tset/test7/tc7", "all");
                    }
                }
            }
        }
    }
}
```

See also

- ‘‘The scenario file’’ in the TETware Programmers Guide.

26. Writing a new API

Question

One of the requirements that I have is to create an API for an internal language we have developed. Can you suggest what are the requirements for creating such an API? The API will allow for remote execution, but distributed execution is not required. Do you have any pointers? What is the magnitude of such an effort?

Answer

When you create a new API for TETware, the first thing to consider is: **Can the new API language be linked to C?**

If the answer is **Yes**, then the task is a relatively simple one. You can provide a glue layer between the new language and the existing C TCM and API library. When you define the way in which test cases are to be called by the TCM, you will probably need to use the dynamic test case interface that is described in Chapter 8 of the TETware Programmers Guide. (If you go down this route, be sure not to use unpublished interfaces in the C TCM or API since these can and do change between TETware releases!)

However, if the answer is **No**, there is a little more work to do. You have to implement your own TCM and API library. If your language is an interpreted one it is probably best to use the (Bourne) Shell API as a starting point. You can find the source code for this below the directory `src/xpg3sh/api` in the TETware distribution.

If you want to see an example of how the Shell API is used, check out the Shell API demonstration test suite which is below the `contrib/SHELL-API` directory in the contrib distribution.

Now some hints as to how to approach the task . . .

1. Interface between `tcc` and the test case.

   a. Command-line arguments.
      `tcc` passes a list of Invocable Components on the command line. Your TCM should use this list to decide which test purpose functions to call. You should be prepared to accept zero or more arguments. Each argument might be:

      — an invocable component number;
      — a range of invocable component numbers (that is: two numbers separated by a hyphen);
      — the word `all` (meaning all invocable components in the test case);
      — a comma-separated list of any of these.

      If no arguments are passed on the command line, your TCM should behave as if `all` had been specified. If the word `all` follows a number or a number range, it means ‘‘all the ICs in the test case beyond the last one specified’’.

      Your TCM should allow for the possibility that an IC might be specified more than once on the command line.

   b. Environment variables.
      `tcc` passes certain information to the test case in the environment. Check out the section entitled ‘‘Communication variables’’ in the TETware Programmers Guide. In particular, you will need to make use of `TET_ACTIVITY`, `TET_CODE` and

TET_CONFIG. You may also need to use TET_ROOT, depending on how your language works.

However, it is also possible for your test case to be invoked directly by a user, so you should allow for the possibility that these environment variables are not set. Look at the Shell API to see how it handles this situation.

c. Current working directory.
When tcc executes a test case, it does so in the **test case execution directory**. The precise location of this directory depends on the settings of certain configuration and environment variables. So you should not make any assumptions about the current working directory when the test case is executed.

d. Execution results file.
The TCM should create a results file called tet_xres in the directory in which it is invoked. tcc reads journal lines from this file when the test case exits.

e. TCM exit status.
Your TCM should exit with a zero status value if no errors occur. If a fatal error occurs you should print a Test Case Manager message describing the problem to the execution results file and exit with a +ve status value.

2. API functions.
As far as possible, your API should (at least) provide functionality of each type that is described in Chapter 11 of the TETware Programmers Guide. Depending on the capabilities of your language, you may need to provide explicit functions to implement the things that the Shell does of itself — these things are described in the last few subsections of Chapter 11. You can refer to the corresponding sections in Chapter 8 for more information if necessary.

One point to note — the description of the Shell version of tet_reason says that the function ‘‘prints a string on the standard output’’. This is because that is the way for a shell function to return a string; the string is picked up in the calling function by using backquotes. When you implement this function in your language, you will probably just want to return the string to the caller.

3. Journal lines.
These are described in Chapter 13 of the TETware User Guide and in Appendix C of the TETware User Guide.

a. Format.
Your API should format and print these lines to the tet_xres file. Be sure to format each line correctly; before printing the line you need to:

— translate embedded newlines to tabs;

— strip trailing white space;

— ensure that the line does not exceed 512 characters.

b. Line types generated by an API.
Your TCM and/or API should generate the following lines at the appropriate times:

- Test Case Manager Start
- Test Purpose Start
- Test Purpose Result

- Invocable Component Start
- Invocable Component End
- Test Case Manager Message
- Test Case Information

All the other line types are generated by `tcc` — they should not be generated by your API.

c. Test purpose results.
You should ensure that each test purpose only generates one Test Purpose Result line, no matter how many times your `tet_result` function is called from each test purpose. See how the Shell API handles this.

See also

- ‘‘Testing structure’’ in the TETware Programmers Guide.
- ‘‘The Test Case Manager’’ in the TETware Programmers Guide.
- ‘‘The C API’’ and ‘‘The Shell and Korn Shell APIs’’ in the TETware Programmers Guide.
- ‘‘Writing a Shell language API-conforming test suite’’ and ‘‘Example Shell API test suite source files’’ in the TETware Programmers Guide.
- ‘‘Running the TETware demonstrations’’ in the TETware User Guide.
- ‘‘Test reporting and journaling’’ in the TETware Programmers Guide.
- ‘‘TETware journal lines’’ in the TETware User Guide.

27. Error message: `tetsyncd: client connection closed`

Question

I am running Distributed TETware. I have a test case that causes the following message to appear on `tcc`'s standard error stream:

```
tetsyncd (syncd.c, 225): client connection closed \
        (sysid = 0, pid = 5749: MTCM)
```

Why does this occur?

Answer

This is a warning message. It means that a process has exited without first logging off from `tetsyncd` (the Synchronisation daemon). I expect that you will see a similar message generated by `tetxresd` as well. (`tetxresd` is the Execution Results daemon.)

The messages show that the process concerned (pid = 5749) is a Test Case or Child Process (MTCM) running on the local system (sysid = 0).

There are two cases to consider:

1. Test case:
   Usually these messages mean that the test case has terminated abnormally.

2. Child process:

   — The process has called one of the `exec()` calls.
   — The process has exited by calling `exit()`.
   — The process has terminated abnormally.

You will need to check the process ID to see whether the process in question is a test case or a child process. If each process prints something to the journal, you can read the process ID from the last five digits of the journal context number.[1] (This is the 3rd subfield in the 2nd field of a line type 520.)

If you determine that the messages are generated as a result of the behaviour of a child process, you will need to see if it is calling one of the `exec()` or `exit()` system calls directly.

Any process that uses the API is supposed to log off the servers before exiting. It doesn't really matter if they don't do this — the servers still behave correctly when the connection goes down unexpectedly — but the warning messages are generated so as to inform you that something unexpected has occurred.

In the case of a child process, the API logs off the servers if the process terminates by returning from `tet_main()`. But if you want to terminate the process by calling `exit()`, you should call `tet_exit()` instead. Finally, if the child process overlays itself by a call to one of the

1. This is not entirely true on a Win32 system. Sometimes more than five digits of context information can appear and the context number in a child process is not usually the same as the process ID. For information about context numbers on Win32 systems, refer to the section which describes context numbers in the appendix entitled ‘‘Implementation notes for TETware on Win32 systems’’ in the TETware User Guide.

`exec()` family of system calls, it should call `tet_logoff()` first.

See also

- The descriptions of `tet_exit()` and `tet_logoff()` in ‘‘The C API’’ in the TETware Programmers Guide.

28. `tccd` can't execute `make` on a remote system

The information in this article is applicable to Distributed TETware.

Question

When I try to build test cases on a remote system, the following error message appears in the journal:

```
110|1 /ts/tc2 15:48:07|Build Start, scenario ref 4-1
50||(exec.c, 136): can't exec make on system 001, \
        server reply code = ER_NOENT
130|1 -1 15:48:08|Build End, scenario ref 4-1
```

It seems that `make` can't be executed on the remote system.

Answer

`tccd` uses the `PATH` environment variable to locate commands in the usual way. On a remote system (that is: a system with a non-zero system ID), the value of `PATH` is inherited from `inetd`. Typically this is set to something like `/bin:/usr/bin`. If `make` doesn't live in one of those places, `tccd` can't find it and prints an error message to this effect.

For example, on Solaris systems, `make` usually lives in `/usr/ccs/bin` and the compiler lives in a directory such as `/opt/SUNWspro/bin`. Clearly these locations are not included in the value of `PATH` that `tccd` inherits from `inetd`.

You can use the `−e` command-line option to pass a different value of `PATH` to `tccd`. For example, on a Solaris system you might modify the `tccd` entry in `/etc/inetd.conf` to look something line this:

```
tcc  stream  tcp  nowait  tet  tet-root/bin/in.tccd  \
        in.tccd -e PATH=/usr/bin:/usr/ccs/bin:/opt/SUNWspro/bin
```

See also

- The `tccd` manual page in the TETware User Guide.
- ‘‘Starting `tccd`’’ in the TETware Installation Guide for UNIX Operating Systems.

29. Does `in.tccd` source `.profile` on a remote system?

The information in this article is not applicable to Win32 systems.

Question

It doesn't look like `in.tccd` sources the `.profile` of the `tccd` user. Is this the case?

Answer

Yes. Since `in.tccd` is not the child of a login shell, environment variables set in a `.profile` or `.login` file are not passed to `in.tccd`.

You can invoke `in.tccd` from within a shell script wrapper if you need this kind of processing. However, if you invoke `in.tccd` from a wrapper, you must ensure that nothing in the shell script touches the standard input. This is because `inetd` passes the client connection to `in.tccd` on the standard input.

See also

- ''`tccd` versions and modes of operation'' in the TETware User Guide.
- ''Starting `tccd`'' in the TETware Installation Guide for UNIX Operating Systems.
- The `tccd` manual page in the TETware User Guide.

30. ER_ERR — the general error code

The architecture used by both TETware-Lite and Distributed TETware consists of several layers. Since Distributed TETware uses a client-server architecture, some of these layers are implemented in client processes whereas other layers are implemented in server processes.

When an error condition occurs, diagnostic messages can be generated by more than one of these layers. Many of these messages are accompanied by a ‘‘server reply code’’ which contains additional information about the cause of the error. Whenever the server reply code is ER_ERR, a more detailed error message will have been generated by a lower-level function. In Distributed TETware this more detailed error message can often be found in the **tccd log file**, which is the place where tccd logs its diagnostic information. The default location for this file is /tmp/tccdlog on a UNIX system or c:/tmp/tccdlog on a Win32 system.

Therefore, when Distributed TETware is used, whenever a Test Case Controller or Test Case Manager message appears in the journal which includes an ER_ERR mnemonic, it is often necessary to look at the tccdlog file in order to determine the cause of the error.

See also

- ‘‘Server reply codes’’, ‘‘tccd log file’’ and the tccd manual page in the TETware User Guide.

31. How to use the dynamic test case interface in the C API

Suppose you have a test case that contains 100 Test Purpose functions.

You must first decide how your Test Purpose functions are to be distributed between the Invocable Components in your test case. (Recall that a test case contains one or more ICs and that each IC can contain one or more TPs; thus an IC is a logical grouping of TP functions.) The simplest way is probably to allocate one TP to each IC. In this case, your test case will contain 100 ICs, numbered 1 to 100. Each IC will contain a single TP.

The TCM uses the interface functions to find out how your test case is organised. First, the TCM calls `tet_getminic()` and `tet_getmaxic()` to determine the IC numbers of the lowest and highest IC, respectively. If your test case contains ICs 1 to 100, you would define these functions as follows:

```
int tet_getminic(void)
{
        return(1);
}

int tet_getmaxic(void)
{
        return(100);
}
```

Then, for each IC number in this range, the TCM calls `tet_isdefic()` to determine whether you have defined a particular IC. Since your test case contains all the ICs in the range 1 to 100, you might define this function as follows:

```
int tet_isdefic(int icnum)
{
        if (icnum >= tet_getminic() && icnum <= tet_getmaxic())
                return(1);
        else
                return(0);
}
```

When the TCM is about to execute the TP function(s) in each IC, it calls `tet_gettpcount()` to determine the number of TPs in each IC. Since your test case is to be organised with just one TP in each IC, you might define this function as follows:

```
int tet_gettpcount(int icnum)
{
        if (tet_isdefic(icnum))
                return(1);
        else
                return(0);
}
```

Before the TCM executes each TP, it needs to set the global API variable `tet_thistest` to the absolute test number within the test case. The TCM calls `tet_gettestnum()` to do this. Since your test case is to be organised with just one TP in each IC and since the IC numbers start at one, the absolute test number is the same as the IC number. So you might define this function as follows:

```
int tet_gettestnum(int icnum, int tpnum)
{
        if (tet_isdefic(icnum) && tpnum == 1)
                return(icnum);
        else
                return(0);
}
```

Finally, the TCM calls `tet_invoketp()` to invoke each test purpose function. Since your test case is to be organised with just one TP in each IC, the function number to execute is the same as the IC number. So you might define this function as follows:

```
int tet_invoketp(int icnum, int tpnum)
{
        if (tpnum != 1) {
                tet_printf("TP %d within IC %d not allowed for \
                        in tet_invoketp()", tpnum, icnum);
                return(0);
        }

        switch (icnum) {
        case 1:
                t001();
                break;
        case 2:
                t002();
                break;
        ...

        case 99:
                t099();
                break;
        case 100:
                t100();
                break;
        default:
                tet_printf("IC %d not allowed for in tet_invoketp()",
                        icnum);
                break;
        }

        return(0);
}
```

The diagnostic messages are not mandatory but might be useful to detect an inconsistency between the code in `tet_invoketp()` and the test case layout information that is returned by the other interface functions.

See also

- ‘‘Test case structure and management’’ and ‘‘The Test Case Manager’’ in the TETware Programmers Guide.
- ‘‘Test suite structure’’ in the TETware User Guide.

32. How to run test cases in a clean environment

Question

We want to be able to run test cases without being influenced by the environment that is in effect when `tcc` is invoked.

Answer

You can use a shellscript exec tool to execute the test case in a pristine environment. The tool must take care not to remove TETware communication variables from the environment, otherwise the TCM/API will be unable to function correctly.

For example:

```
#!/bin/sh
#
# exec tool which executes a test case in a known, pristine environment
#

tcname=${1:?}
shift

exec env - \
        ${PATH+PATH="$PATH"} \
        ${TZ+TZ="$TZ"} \
        ${TET_ACTIVITY+TET_ACTIVITY="$TET_ACTIVITY"} \
        ${TET_CODE+TET_CODE="$TET_CODE"} \
        ${TET_CONFIG+TET_CONFIG="$TET_CONFIG"} \
        ${TET_EXECUTE+TET_EXECUTE="$TET_EXECUTE"} \
        ${TET_ROOT+TET_ROOT="$TET_ROOT"} \
        ${TET_RUN+TET_RUN="$TET_RUN"} \
        ${TET_SUITE_ROOT+TET_SUITE_ROOT="$TET_SUITE_ROOT"} \
        ${TET_TMP_DIR+TET_TMP_DIR="$TET_TMP_DIR"} \
        ${TET_TIARGS+TET_TIARGS="$TET_TIARGS"} \
        ${TET_TSARGS+TET_TSARGS="$TET_TSARGS"} \
        ./$tcname $*
```

When a process is executed by the `env(1)` command with the – option, it is necessary to pass all the required environment variables explicitly on the command line. Other variables which are inherited by the exec tool from `tcc` and/or `tccd` are not made available to the test case.

The `${`*variable*`+`*word*`}` syntax is used to avoid passing an empty variable to the test case if the corresponding variable was not defined in the parent environment.

On some systems you may need to add other variables to the list. For example: `HOME` and `LOGNAME`; variables that might be needed by a dynamic linking scheme such as `LD_LIBRARY_PATH`, `LIBPATH` or `SHLIB_PATH`; anything to do with locales; and so forth. Be sure to include `SystemRoot` on a Win32 system, otherwise the Windows Socket library won't work.

See also

- ‘‘Communication variables’’ in the TETware Programmers Guide.

33. How to copy executed commands output to the journal

Question

I have some test cases that use the C API. As part of my test program, I need to execute a shell script using `system()`. I would like the output of shell script to go into the journal file.

I tried setting `TET_OUTPUT_CAPTURE=True` in my `tetexec.cfg` file and ran the test. It just hangs. Is there anything else that I need to do to solve this problem?

Is there any other way other than by setting `TET_OUTPUT_CAPTURE`?

Answer

There are a couple of ways that you can do this — a simple way where `tcc` does the output capture for you, and a more complicated way where you do the output capture within the test case.

Method 1 — have `tcc` do the work for you

Recall that the value of `TET_OUTPUT_CAPTURE` provides a default value for `TET_API_COMPLIANT`. The value of `TET_API_COMPLIANT` must be True if your test case uses a TETware API, otherwise `tcc` processes your test case as a non API-conforming test case.

So if you want to run an API-conforming test case with output capture mode enabled, you must set both of these variables explicitly, thus:

```
TET_OUTPUT_CAPTURE=true
TET_API_COMPLIANT=true
```

When you do this, `tcc` captures all the output from each test case and inserts it in the journal before any of the API-generated output from the test case. The API doesn't indicate which section of the output comes from each test purpose function.

Method 2 — do the output capture within the test case

If you want output from your shell script to appear in the journal between the appropriate TP Start and TP Result lines, you will need to redirect your shell script's `stdout` and `stderr` to a file, then collect the contents of this file in your test code and use `tet_infoline()` to print it to the journal. When you do this, you don't need to set `TET_OUTPUT_CAPTURE=true`.

For example, the following function will do this on a UNIX system:

```
int system_with_capture(cmd)
char *cmd;
{
        static char template[] = "/tmp/capXXXXXX";
        char capfile[sizeof template];
        char buf[BUFSIZ];
        FILE *fp;
        char *p;
        int rc;

        (void) strcpy(capfile, template);
        (void) mktemp(capfile);

        (void) sprintf(buf, "%s > %s 2>&1", cmd, capfile);
        (void) unlink(capfile);
        if ((rc = system(buf)) == 127 || rc == -1)
                return(rc);

        if ((fp = fopen(capfile, "r")) == (FILE *) 0)
                tet_printf("can’t open output capture file %s, \
                        errno = %d", capfile, errno);
        else {
                tet_printf("output from \"%s\":", cmd);
                tet_infoline("----------------------------------");
                while (fgets(buf, sizeof buf, fp) != (char *) 0) {
                        for (p = buf; *p; p++)
                                if (*p == ’\n’)
                                        *p = ’\0’;
                        tet_infoline(buf);
                }
                tet_infoline("----------------------------------");
                (void) fclose(fp);
                (void) unlink(capfile);
        }

        return(rc);
}
```

Which method you choose depends on how important it is for the captured output to be identified with a particular test purpose function.

See also

- ‘‘Execute mode processing’’ and ‘‘Configuration variables which modify TETware’s operation’’ in the TETware Programmers Guide.
- ‘‘Making journal entries’’ in each chapter that describes API functions in the TETware Programmers Guide.

34. How to run individual Invocable Components from the `tcc` command line

Question

Is it possible to instruct `tcc` to run individual ICs from the command line without having to make a `/tc/test1{1}` type of entry in the `tet_scen` file? This would be really handy for debugging parts of a test.

Answer

You can do this using `tcc`'s `–l` command-line option. When you specify a scenario line using `–l`, the `tet_scen` file is not read (unless it is specified explicitly with the `–s` option).

For example:

```
tcc -e -l /tc/test1{1} ...
```

See also

- The `tcc` manual page in the TETware User Guide.

35. How to identify test case timeouts in the journal file

Question

When `tcc` is invoked with the `–t` option, the journal reports the test purpose result for a timeout condition as `NORESULT (auto-generated by TCC)`. How can I determine that the cause of this failure was a test case timeout?

Answer

The fact that `tcc` has timed out a test case or tool is indicated by the exit status in the Build End, Test Case End or Clean End line in the journal. The exit status appears in the 2nd subfield of the 2nd field on lines of type 130, 80 and 320. A timeout is indicated by a value of -2 in this field.

A zero or +ve exit status in the End line comes from the test case or tool, while -ve exit status indicates some condition that is reported by `tcc`. Your report writer should examine this field and generate a suitable message when the exit status value in the End line is non-zero. For the convenience of C language report writers, symbolic constants which relate to journal values are provided in the file *tet-root*`/inc/tet3/tet_jrnl.h`.

Users of Win32 systems are advised not to use `tcc –t` to time out test cases and tools.

See also

- ‘‘Journal line parameters" in the TETware Programmers Guide.
- ‘‘Test case termination’’ in the ‘‘Implementation notes for TETware on Win32 systems’’ in the TETware User Guide.
- ‘‘TETware journal lines’’ in the TETware User Guide.
- The `tcc` manual page in the TETware User Guide.

36. Running different randomly selected test cases in parallel on several systems

Question

The scenario we would like to setup takes a set of test cases, selects a random element within an outer repeat loop. This works fine locally. But when we wrap this scenario in a `:remote:` directive, what we get is the same test case being processed on all remote machines, rather than the whole scenario being executed on all remote machines.

I want to be able to have a different (randomly selected) test case executed on each remote node. The goal is to setup a stress test environment to simulate a lot of concurrent operations, not a lot of simultaneous identical operations.

Answer

The behaviour that you observe is correct. Each test case within the scope of a `:remote:` directive (which doesn't include system 0) is processed on each of the specified systems at the same time — effectively ‘‘in parallel’’.

If the `:remote:` directive encloses a `:random:` directive, the `:random:` directive selects a test case, then the `:remote:` directive processes the test case on each of the named systems.

The trick is to push the system attribute further up the scenario tree. For example, a scenario that looks something like this will do it:

```
all
        :parallel:^systems

systems
        ^sys1
        ^sys2
        ^sys3
        ^sys4
        ^sys5

sys1
        :remote,1:^loop

sys2
        :remote,2:^loop

sys3
        :remote,3:^loop

sys4
        :remote,4:^loop

sys5
        :remote,5:^loop

loop
        :repeat,10;random:^tclist

tclist
        /ts/tc1/tc1
        /ts/tc2/tc2
        /ts/tc3/tc3
        /ts/tc4/tc4
```

```
/ts/tc5/tc5
...
```

This scenario must be run in ETET compatibility mode. (that is: set `TET_COMPAT=etet` in the per-mode configuration(s)) The scenario relies on the way that `tcc` inserts **implied sequential** directives into the tree when processing a scenario in ETET mode.

See also

- The description of the `:parallel:` directive in the TETware Programmers Guide.

37. How do environment variables get passed to test cases?

Question

When we run using TET-Lite, we pick up environment information by sourcing files specify environment variables such as `TET_ROOT`, `TET_SUITE_ROOT` and `PATH`. At what point does this information get put into the environment on the remote machine when Distributed TETware is used?

Answer

When `tcc` processes a test case, it puts certain variables in the test case's environment. These variables are called **communication variables** and are used by `tcc` to communicate information to the TCM and API. `tcc` always puts all of these variables in the test case's environment, supplying default values if necessary. This operation is performed by both the Lite and the Distributed versions of `tcc`.

The values of some of these variables on the local system are derived from environment variables that you provide to `tcc` (namely: `TET_ROOT`, `TET_EXECUTE`, `TET_SUITE_ROOT` and `TET_RUN`). By contrast, when `tcc` puts communication variables in the environment of a test case or tool on a remote system, it derives values for these variables from the values that you specify in `tetdist.cfg`.

The values of other communication variables (such as `TET_CONFIG` and `TET_ACTIVITY`) are generated internally by `tcc`.

In TETware-Lite, test cases and tools are children of `tcc` so they inherit environment variables from `tcc`.

By contrast, in Distributed TETware, test cases and tools are children of `tccd`, so they inherit environment variables from `tccd`. `tcc` sends a copy of its environment to `tccd` on system 0 (the local system) soon after logging on. This ensures that test cases and tools running on the local system receive (almost) the same environment as they would if run by the Lite version of `tcc`. (In practice, the environment inherited by test cases and tools running on the local system is the union of `tcc`'s and `tccd`'s environment, with the values of variables supplied by `tcc` having precedence over any corresponding values in `tccd`'s environment.)

However, `tcc` does `not` send a copy of its environment to tccd on other systems; it only sends the communication variables as described previously. This means that the values of other variables (such as `PATH`) are the ones that are in force at the time that `tccd` starts up (although additional variables may be specified using the `–e` command-line option). In particular, the **inetd** version of `tccd` inherits its environment from `inetd`, which is usually fairly minimal.

See also

- ''Initial processing'' and ''Communication variables'', both in the chapter entitled ''The Test Case Controller'' in the TETware Programmers Guide.
- The `tccd` manual page in the TETware User Guide.

38. How to process a scenario that contains test cases that use different APIs

Question

Is it possible to define a scenario that contains test cases that use different APIs? In particular, Java test cases must be executed by a Java-specific exec tool which cannot execute test cases that use other APIs.

Answer

`tcc` is able to process scenarios that contain a mixture of test cases. The issue to be resolved is how to arrange for the build, exec and clean tools to recognise the different types of test case and process them accordingly.

In build and clean mode it is probably best to provide a makefile to build each test case. The Java build tool could be invoked from the makefiles that build the Java test cases.

In exec mode, one way to handle this is to have a top-level exec tool which switches on the name of the test case being executed and invokes an appropriate language-specific exec tool. Of course, only the Java exec tool is mandatory — the others are optional.

If you do this you need to organise the test suite in a way that the exec tool understands. For example, you might define a directory hierarchy which looks like this:

```
tset/
        c-tests/
                ...
        shell-tests/
                ...
        perl-tests/
                ...
        java-tests/
                ...
```

Then you could use a top-level exec tool that looked something like this:

```
#!/bin/sh
# top level exec tool which may invoke a language-specific exec tool
# depending on test case name

# extract the name(s) of the language-specific exec tool(s) and their
# optional argument(s) from the config file
C_EXEC_TOOL=
C_EXEC_FILE=
SHELL_EXEC_TOOL=
SHELL_EXEC_FILE=
PERL_EXEC_TOOL=
PERL_EXEC_FILE=
JAVA_EXEC_TOOL=
JAVA_EXEC_FILE=
eval "‘sed -n ’s/#.*//
        /^[     ]*\$/d
        /^[A-Z]*_EXEC_[A-Z]*=/s/\([^=]*\)=\(.*\)/\1="\2"/p’ ${TET_CONFIG:?}‘"
```

```
# determine the name of the test case to execute
tcname=${1:?}
shift

# work out if we can execute the test case directly
# or should invoke a language-specific exec tool and optional
# tool argument
cmd=
case $tcname in
*/c-tests/*)
	if test -z "$C_EXEC_TOOL"
	then
		cmd=./$tcname
	else
		cmd="$C_EXEC_TOOL $C_EXEC_FILE $tcname"
	fi
	;;
*/shell-tests/*)
	if test -z "$SHELL_EXEC_TOOL"
	then
		cmd=./$tcname
	else
		cmd="$SHELL_EXEC_TOOL $SHELL_EXEC_FILE $tcname"
	fi
	;;
*/perl-tests/*)
	if test -z "$PERL_EXEC_TOOL"
	then
		cmd=./$tcname
	else
		cmd="$PERL_EXEC_TOOL $PERL_EXEC_FILE $tcname"
	fi
	;;
*/java-tests/*)
	if test -z "$JAVA_EXEC_TOOL"
	then
		echo "$0: JAVA_EXEC_TOOL is null or not set" 1>&2
		exit 1
	else
		cmd="$JAVA_EXEC_TOOL $JAVA_EXEC_FILE $tcname"
	fi
	;;
*)
	echo "$0: can’t determine test case type: $tcname" 1>&2
	exit 1
	;;
esac

# finally, execute the test case or tool
exec $cmd ${1:+"$@"}
```

In the execute mode configuration you would define `TET_EXEC_TOOL` to be the name of this file. Then if required you could define one or more of `C_EXEC_TOOL`, `C_EXEC_FILE`, `SHELL_EXEC_TOOL`, `SHELL_EXEC_FILE`, `PERL_EXEC_TOOL`, `PERL_EXEC_FILE`, `JAVA_EXEC_TOOL`, and `JAVA_EXEC_FILE` in the execute mode configuration. Only the `JAVA_EXEC_TOOL` is mandatory; the others are optional.

See also

- ‘‘The Test Case Controller’’ and ‘‘Configuration variables which modify TETware’s operation’’ in the TETware Programmers Guide.

39. How to create a new thread in a Java API-conforming test case

Question

I have a test program looks like this:

```
public class test {
        public static void main(String args[]) {
        new testThread().start();
        }
}

class testThread extends Thread {
        public void run()
        {
        // the test program is here.
        }
}
```

How can I convert it to use the TETware Java API?

Answer

When you create a new thread in a program that uses the TETware Java API, instead of saying

```
class testThread extends Thread
```

you need to say

```
class testThread extends TetThread
```

Here is an example of a simple test case that creates a new thread:

```
import java.lang.*;
import TET.*;

/*
 * Test case for threads.
 */
public class ThrTest extends SimpleTestCase
{
        public static final int SLEEP_TIME = 5;

        public static void main(String[] args)
        {
                main("ThrTest", args, new ThrTest());
        }

        public void i1t1(TestSession ts)
        {
                TestThread thread;

                thread = new TestThread(ts, 1, SLEEP_TIME);
                thread.start();
                ts.tet_infoline("This is the parent thread: "
                        + Thread.currentThread());
                ts.tet_infoline("Parent printing child thread: "
```

```
                                + thread);

                        try
                        {
                                thread.join(2 * SLEEP_TIME);
                        }
                        catch (InterruptedException e)
                        {
                                ts.tet_infoline("Caught exception \
                                        while joining child thread:");
                                ts.tet_infoline(e.toString());
                                ts.tet_result(ts.TET_UNRESOLVED);
                        }

                        ts.tet_infoline("parent thread returns");
                        ts.tet_result(ts.TET_PASS);
                }
        }

        class TestThread extends TetThread
        {
                private TestSession ts;
                private int sleepTime;

                TestThread(TestSession ts, int id, int sleepTime)
                {
                        super(ts, "TestThread() #" + id, sleepTime * 1000L);
                        this.ts = ts;
                        this.sleepTime = sleepTime;
                }

                public void run()
                {
                        ts.tet_infoline("Child thread ("
                                + Thread.currentThread()
                                + "): starting to run");

                        try
                        {
                                sleep(sleepTime * 1000L);
                        }
                        catch (InterruptedException e)
                        {
                                ts.tet_infoline("Child thread ("
                                        + Thread.currentThread()
                                        + ") caught exception while sleeping: ");
                                ts.tet_infoline("       " + e);
                                ts.tet_result(ts.TET_UNRESOLVED);
                        }

                        ts.tet_infoline("Child thread ("
                                + Thread.currentThread()
                                + ") exits");
                }
        }
```

Here is the journal output when this test case is run:

```
0|3.4ea 12:38:06 19990928|User: tet (106) TCC Start, \
        Command line: tcc -epl /ts/tc1/ThrTest
5|Linux fir 2.0.35 #6 Sat Mar 13 09:21:49 GMT 1999 i686| \
        Local System Information
20|/home/tet/jtests/tetexec.cfg 1|Config Start
30||TET_EXPAND_CONF_VARS=true
30||TET_OUTPUT_CAPTURE=true
30||TET_API_COMPLIANT=true
30||TET_EXEC_IN_PLACE=true
30||TET_EXEC_TOOL=ksh
30||TET_EXEC_FILE=/home/tet/jtests/exectool.ksh
30||TET_JAVA_PATH=/usr/local/java/bin/java
30||TET_PASS_TC_NAME=True
30||TET_VERSION=3.4ea
40||Config End
10|0 /ts/tc1/ThrTest 12:38:06|TC Start, scenario ref 1-0
15|0 3.4ea 1|TCM Start
400|0 1 1 12:38:06|IC Start
200|0 1 12:38:06|TP Start
520|0 1 00030029 1 1|This is the parent thread: \
        Thread[main,5,main]
520|0 1 00030029 1 2|Parent printing child thread: \
        Thread[TestThread() #1,5,TET:ThrTest:IC1TP1]
520|0 1 00030029 1 3|parent thread returns
520|0 1 00030029 2 1|Child thread \
        (Thread[TestThread() #1,5,TET:ThrTest:IC1TP1]): \
        starting to run
520|0 1 00030029 2 2|Child thread \
        (Thread[TestThread() #1,5,TET:ThrTest:IC1TP1]) exits
220|0 1 0 12:38:11|PASS
410|0 1 1 12:38:11|IC End
80|0 0 12:38:12|TC End, scenario ref 1-0
900|12:38:12|TCC End
```

See also

- ‘‘Using Threads’’ in the chapter that describes the Java API in the TETware Programmers Guide.

40. How to run TETware from an NFS server

The information in this article is not applicable to Win32 systems.

You can run TETware from a server over NFS.

Here are some instructions which describe how to set up one of the configurations that we support when we run the TETware training course:

1. Create a user called `tet` in group `tet` on the server.
2. Install TETware on the server in a directory owned by the user `tet`; for example, in /home/tet.

   In the instructions that follow, assume that the server is called `tetserver`.
3. Have each workstation mount `tetserver:/home/tet`.
   For example:

   ```
   mkdir -p /nfs/tetserver/home/tet
   mount -r tetserver:/home/tet /nfs/tetserver/home/tet
   ```

4. Create a `tet root` directory on each workstation in some private area.
   For example, for user `fred`:

   ```
   mkdir /home/fred/tet3
   chgrp tet /home/fred/tet3
   chmod g+ws /home/fred/tet3
   TET_ROOT=/home/fred/tet3
   export TET_ROOT
   ```

   It is best to put these last two lines in each user's `.profile`.
5. Create symbolic links in each private **tet root** directory to the server's **tet root** directory.
   For example:

   ```
   cd /home/fred/tet3
   ln -s /nfs/tetserver/home/tet/bin .
   ln -s /nfs/tetserver/home/tet/inc .
   ln -s /nfs/tetserver/home/tet/lib .
   ```

6. Add the group `tet` to each user's supplementary group list.
7. Have each user work with a umask of 2.
8. In Distributed TETware, run `in.tccd` with the `–m2` option.

See also

- The `tccd` manual page in the TETware User Guide.

41. Configuration variable exchange errors in Distributed TETware

Question

When I try to execute the test case as specified in `tet_scen` I get the following error in the journal file:

```
0|3.2 15:02:52 19980805|User: tet TCC Start, Command line: tcc -e
5|Windows_NT phobos 4 0 586|Local System Information
50||(tcconf.c, 258): unexpected STCC reply code ER_ERR
50||(tcxconf.c, 218): tet_tcsndconfv failed, rc = ER_ERR
50||(config.c, 988): tet_tcxconfig() failed when performing \
        exec mode configuration variable exchange with system 1: \
        rc = ER_ERR
900|15:02:52|TCC End
```

Answer

Whenever you see a server reply code of `ER_ERR` which relates to an operation performed by `tccd`, a more detailed error message which describes the cause of the problem is printed in the tccdlog file on the remote system.

The most common cause of a ‘‘configuration variable exchange’’ error is that `tccd` on the remote machine could not find the configuration file. You need to provide configuration files for each of the selected modes of operation on the remote systems as well as on the local system.

When you run `tcc` in execute mode: if the test suite root directory on system 1 is called `/home/tet/tests` (as defined in the `tetdist.cfg` file on the local system), `tccd` on system 1 looks for a file called `/home/tet/tests/tetexec.cfg` when `tcc` performs the configuration variable exchange with system 1. This file must exist, even if it is empty.

See also

- ‘‘The Test Case Controller’’ and ‘‘Configuration variable processing in Distributed TETware’’ in the TETware Programmers Guide.
- ‘‘`tccd` log file’’ in the TETware User Guide.

42. Error message: `tccd: refused login request from` *hostname*

Question

When I run the Distributed `tcc`, it is unable to connect to `tccd`. The following message appears in the tccdlog file on the system in question:

```
tccd (102) 8 Jul 15:26:25: connection received from phobos
tccd (102) 8 Jul 15:26:25 (tccd_in.c, 435): refused login request \
        from phobos
tccd (102) 8 Jul 15:26:25 (tccd.c, 398): client connection closed \
        (sysid = 0, pid = 140: MTCC)
```

Answer

This means that `tcc` on the system called `phobos` tried to log on to `tccd`. `tccd` looked up this name in the `systems.equiv` file but didn't find it. So the logon request was refused.

You should check that the `systems.equiv` file contains an entry for each system from which you want `tccd` to accept connections.

Each entry in this file should be whatever host name is returned when `gethostbyaddr()` is called with the system's IP address as argument. If your system is running a name server, this will usually be a fully-qualified domain name. If a call to `gethostbyaddr()` can't resolve the IP address, you can specify the IP address in the `systems.equiv` file instead.

See also

- The `systems.equiv` manual page in the TETware User Guide.
- ‘‘The systems equivalence file’’ in the TETware Installation Guide.