UTL_FILE

UTL_FILE is a package that is supplied to allow PL/SQL to read and create text files in the file system of the server. The keywords here are:

- **Text Files** – UTL_FILE can only read and create clear text files. Specifically, it cannot be used to read or create binary files. Special characters contained within arbitrary binary data will cause UTL_FILE to do the wrong thing.

- **File System of the Server** – UTL_FILE can only read and write to the file system of the database server. It cannot read or write to the file system the client is executing on if that client is not logged onto the server itself.

UTL_FILE is an appropriate tool for creating reports and flat file dumps of data from the database or for reading files of data to be loaded. In fact, if you refer to Chapter 9 on *Data Loading* in this book, we have a full blown example of using UTL_FILE to create flat file dumps in a format suitable for easy reloading. UTL_FILE is also a good choice for doing 'debugging'. If you refer to the Chapter 21 on *Fine Grained Access Control* we introduced the DEBUG package. This package makes heavy use of UTL_FILE to record messages in the file system.

UTL_FILE is an extremely useful package as long as you are aware of its limits. If not, you may attempt to use UTL_FILE in a way in which it will not work correctly (it might work in testing but not in production) leading to frustration. Like all tools, knowing its limits and how it works will be useful.

We'll look at some issues frequently encountered when using `UTL_FILE` including:

- The `UTL_FILE_DIR` parameter in `init.ora`.
- Accessing mapped drives (network drives) in a Windows environment (there are no related issues in a Unix environment).
- Handling exceptions from `UTL_FILE`
- Using `UTL_FILE` to create static web pages on a recurring basis.
- The infamous 1023 byte limit.
- Getting a directory listing so you can process all files in a directory.

The UTL_FILE_DIR init.ora parameter

This is a key part to using `UTL_FILE`, which always runs as the Oracle software owner – it is your dedicated server or shared server that is performing the I/O and these are running as 'Oracle'. Given that they run as Oracle and Oracle can read and write to its datafiles, configuration files, and so on – it would be a very bad thing if `UTL_FILE` permitted access to just any directory. The fact that we must explicitly set the directories we want to be able to write to in the `init.ora` is a great safety feature – it is not an encumbrance. Consider if `UTL_FILE` allowed you to write to any directory Oracle could – any user could then use `UTL_FILE.FOPEN` to rewrite your system datafile. That, to put it mildly, would be a bad thing. Therefore, your DBA must open up access to specific directories – explicitly. You cannot even specify a root directory and allow access to it and all directories underneath – you must explicitly list out each and every directory you want to read and write to with `UTL_FILE`.

It should be noted that this `init.ora` parameter is not changeable while the database is up and running. You must restart the instance in order for a directory entry to be added or removed.

The `UTL_FILE_DIR` `init.ora` parameter takes one of two forms:

```
utl_file_dir = (c:\temp,c:\temp2)

or

utl_file_dir = c:\temp
utl_file_dir = c:\temp2
```

That is, you may either use a list of comma separated directories enclosed in parenthesis or you may list each directory on a line one after the other. The keywords here being 'one after the other'. If you have the following as the last lines in your `init.ora` file:

```
utl_file_dir = c:\temp
timed_statistics=true
utl_file_dir = c:\temp2
```

Only the **last** entry for `UTL_FILE_DIR` will be used. The first directory entry will effectively be ignored. This can be quite confusing as there will be no warning message or `alert.log` entries indicating the first `UTL_FILE_DIR` entry is ignored. All `UTL_FILE_DIR` entries must be contiguous in the `init.ora` file.

One word of warning on the Windows Platform with regards to this `init.ora` parameter. If you decide to add the trailing \ to the `UTL_FILE_dir` parameter like this:

```
utl_file_dir = c:\temp\
utl_file_dir = c:\temp2
```

You will receive the following error message upon startup:

```
SVRMGR> startup
LRM-00116: syntax error at 'c:\temputl_file_' following '='
LRM-00113: error when processing file 'C:\oracle\admin\tkyte816\pfile\init.ora'
ORA-01078: failure in processing system parameters
```

That is because the \ is considered an escape character at the end of the line in the `init.ora` file. It would allow you normally to continue long entries on 2 lines. You must simply use two slashes:

```
utl_file_dir = c:\temp\\
utl_file_dir = c:\oracle
```

in order to avoid this concatenation.

Another closing note on this `init.ora` parameter. If you use a trailing slash in the `init.ora`, you must use the trailing slash in your `fopen` calls. If you omit the trailing slash in the `init.ora`, you should omit it in your `fopen` calls as well. The directory parameter to `fopen` should match in case and contents the value you put into the `init.ora` file.

Accessing Mapped Windows Drives

This is a common area of confusion, especially to people used to working with Unix. On Unix, if you mount a device (for example, NFS mount a remote disk) – it is immediately visible to everyone on that machine – regardless of their session. Each user may have different access rights to it, but the mounted disk is an attribute of the system and not a specific session.

In Windows, this is very different. I may have many sessions executing on a given server and each will have its own set of 'disk drives' that it sees. It may very well be that when I log onto a machine – I see a network resource 'disk `D:`' that belongs physically to some other machine. That does not mean that every process running on that machine can see that disk. This is where the confusion comes in.

Many people log into the server and see 'disk `D:`'. They configure the `init.ora` to have `UTL_FILE_dir = d:\reports` in it ; a directory to write reports to using `UTL_FILE`. At runtime however they receive:

```
ERROR at line 1:
ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at "SYS.UTL_FILE", line 98
ORA-06512: at "SYS.UTL_FILE", line 157
```

Which if they used an exception handler (see below for the one I like to use) they would see something more informative like:

```
ERROR at line 1:
ORA-20001: INVALID_PATH: File location or filename was invalid.
ORA-06512: at "TKYTE.CSV", line 51
ORA-06512: at line 2
```

Well, as far as they can tell – `D:\reports` is just fine. They use explorer and it is there. They use a DOS window and it is there. Only Oracle doesn't seem to be able to see it. This is because when the system started – `D:` drive didn't exist and furthermore, the account under which Oracle is running by default cannot see any network resources. Try as hard as you like, mounting that disk in any way possible, Oracle will not 'see' it.

When an Oracle instance is created the services that support it are setup to 'Log On As' the `SYSTEM` (or operating system) account, this account has very few privileges and no access to Windows NT Domains. To access another Windows NT machine the `OracleServiceXXXX` must be setup to logon to the appropriate Windows NT Domain as a user who has access to the required location for `UTL_FILE`.

To change the default logon for the Oracle services, go to (in Windows NT):

Control Panel | Services | OracleServiceXXXX | Startup | Log On As; (where XXXX is the instance name)

In Windows 2000, this would be:

Control Panel | Administrative Tools | Services | OracleServiceXXXX | Properties | Log On Tab; again XXXX is the instance name)

Choose the This Account radio button, and then complete the appropriate domain login information. Once the services have been setup as a user with the appropriate privileges, there are two options for setting `UTL_FILE_DIR`:

- Mapped Drive: To use a mapped drive, the user that the service starts as must have setup a drive to match `UTL_FILE_DIR` and be logged onto the server when `UTL_FILE` is in use.
- Universal Naming Convention: UNC is preferable to Mapped Drives because it does not require anyone to be logged on and `UTL_FILE_DIR` should be set to a name in the form `\\<machine name>\<share name>\<path>`.

You will of course need to stop and restart Oracle after changing the properties of the service.

Handling Exceptions

`UTL_FILE` throws exceptions when it encounters an error. Unfortunately, it uses user-defined exceptions – exceptions it has defined in its package specification. These exceptions, if not caught by name, produce the following less then useful error message:

```
ERROR at line 1:
ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at "SYS.UTL_FILE", line 98
ORA-06512: at "SYS.UTL_FILE", line 157
```

This tells you nothing about the error itself. In order to solve this issue, we have to surround our calls to UTL_FILE with an exception block that catches each of the exceptions by name. I prefer to then turn these exceptions into RAISE_APPLICATION_ERROR exceptions. This allows me to assign an ORA- error code and supply a more meaningful error message. We used this in the preceding example to turn the above error message into:

```
ORA-20001: INVALID_PATH: File location or filename was invalid.
```

Which is much more useful. The block I always use for this is:

```
exception
    when utl_file.invalid_path then
        raise_application_error(-20001,
       'INVALID_PATH: File location or filename was invalid.');
    when utl_file.invalid_mode then
        raise_application_error(-20002,
      'INVALID_MODE: The open_mode parameter in FOPEN was
       invalid.');
    when utl_file.invalid_filehandle then
        raise_application_error(-20002,
        'INVALID_FILEHANDLE: The file handle was invalid.');
    when utl_file.invalid_operation then
        raise_application_error(-20003,
       'INVALID_OPERATION: The file could not be opened or
        operated on as requested.');
    when utl_file.read_error then
        raise_application_error(-20004,
       'READ_ERROR: An operating system error occurred during
        the read operation.');
    when utl_file.write_error then
        raise_application_error(-20005,
            'WRITE_ERROR: An operating system error occurred
             during the write operation.');
    when utl_file.internal_error then
        raise_application_error(-20006,
            'INTERNAL_ERROR: An unspecified error in PL/SQL.');
end;
```

I actually keep this in a small file and read it into each routine that uses UTL_FILE to catch the exception and 'rename it' for me.

Dumping a Web Page to Disk

This is such a frequently asked question; I thought I would include it here. The scenario is that you are using Oracle WebDB, Oracle Portal, or have some procedures that use the Web Toolkit (the htp packages). You would like to take a report one of these tools is capable of displaying and instead of dynamically generating that report for each and every user – you would like to create a static file with this report every X minutes or hours. This is in fact how I myself generate the home page of the web site I manage at work. Every 5 minutes we regenerate the home page with dynamic data instead of generating each time for the thousands of hits we get during that period of time. Cuts way down on the resources needed to get the home page served up. I do this for frequently accessed, dynamic pages where the underlying data is relatively static (slow changing).

The following procedure is a generic procedure that I use for doing this:

```
create or replace procedure dump_page( p_dir     in varchar2,
                                       p_fname   in varchar2 )
is
    l_thePage       htp.htbuf_arr;
    l_output        utl_file.file_type;
    l_lines         number default 99999999;
begin
    l_output := utl_file.fopen( p_dir, p_fname, 'w', 32000 );

    owa.get_page( l_thePage, l_lines );

    for i in 1 .. l_lines loop
        utl_file.put( l_output, l_thePage(i) );
    end loop;

    utl_file.fclose( l_output );
end dump_page;
/
```

It is that simple. We only have to open a file, get the HTML page, print each line, and close the file. If I call this after calling a WebDB procedure – it will save the output of that WebDB procedure to the file I name.

The only caveat here is that we will be running the WebDB procedure not from the web but directly. If any of the code in the WebDB procedures uses the CGI environment, then that procedure will fail – since the environment was not set up. We can solve that simply by using a small block of code to setup an environment for our WebDB routine:

```
declare
    nm  owa.vc_arr;
    vl  owa.vc_arr;
begin
    nm(1) := 'SERVER_PORT';
    vl(1) := '80';
    owa.init_cgi_env( nm.count, nm, vl );
    -- run your webdb procedure here
    dump_page( 'directory', 'filename' );
end;
/
```

For example, if our WebDB code wanted to verify it was being run from the server on port 80 – we would need to provide the above environment for it. You would add any other environment variables that are relevant to your application in this block

Now all you need to do is refer to the `DBMS_JOB` section and schedule this block of code to be executed on whatever cycle you need.

1023 Byte Limit

Once upon a time, there was a 1023 byte limit to UTL_FILE. Each line written to a file could not exceed 1023 bytes. If it did, an exception was raised and UTL_FILE would fail. Fortunately, in Oracle 8.0.5, they introduced an overloaded version of FOPEN that allows us to specify the maximum line length up to 32 KB in size. 32 KB is the largest size a PL/SQL variable can ever be and is typically sufficient for most purposes.

Unfortunately, the documentation has this newly overloaded FOPEN function documented many pages *after* the original function. This leads to many people overlooking this ability. I still get many questions on this today with version 8.1.7. People did not see the overloaded version of FOPEN; they hit the limit and need to know how to get around it. The answer is simple – but you have to read through every UTL_FILE function to find it easily!

The solution to this particular problem is to use UTL_FILE in the way I did in the previous DUMP_PAGE routine above. The fourth parameter to UTL_FILE.FOPEN is the maximum length of the line of text you would like to produce. In my case above, I allow for up to 32,000 bytes per line.

Reading A Directory

This is a missing piece of functionality in the UTL_FILE package. Frequently people want to setup a recurring job that will scan a directory for new input files and process them, perhaps loading the data into the database. Unfortunately, out of the box there is no way for PL/SQL to read a directory listing. We can however use a tiny bit of Java to give us this ability. The following example demonstrates how you might accomplish this.

First I create a user with the minimum set of privileges needs to perform this operation and to be able to list files in the /tmp directory. If you wish to read other directories, you would need to make more calls to dbms_java.grant_permission (see Chapter 19 on *Java Stored Procedures* for more information) or change the /tmp into * to provide the ability to list all directories.

```
SQL> connect system/manager

system@DEV816> drop user dirlist cascade;
User dropped.

system@DEV816> grant create session, create table, create procedure
  2  to dirlist identified by dirlist;
Grant succeeded.

system@DEV816> begin
  2      dbms_java.grant_permission
  3      ('DIRLIST',
  4       'java.io.FilePermission',
  5           '/tmp',
  6           'read' );
  7  end;
  8  /
PL/SQL procedure successfully completed.
```

Next, after connecting as this new user DirList, we set up a global temporary table in this schema (to hold the directory listing). This is how we will get the results from the Java stored procedure back to the caller – in the temporary table. We could have used other means (strings, arrays and such) as well.

```
SQL> connect dirlist/dirlist
Connected.

dirlist@DEV816> create global temporary table DIR_LIST
  2  ( filename varchar2(255) )
  3  on commit delete rows
  4  /
Table created.
```

Now we create a Java stored procedure to do the directory listing. For ease of programming, I am using SQLJ to avoid having to code lots of JDBC calls:

```
dirlist@DEV816> create or replace
  2     and compile java source named "DirList"
  3  as
  4  import java.io.*;
  5  import java.sql.*;
  6
  7  public class DirList
  8  {
  9  public static void getList(String directory)
 10                     throws SQLException
 11  {
 12      File path = new File( directory );
 13      String[] list = path.list();
 14      String element;
 15
 16      for(int i = 0; i < list.length; i++)
 17      {
 18          element = list[i];
 19          #sql { INSERT INTO DIR_LIST (FILENAME)
 20                 VALUES (:element) };
 21      }
 22  }
 23
 24  }
 25  /
Java created.
```

The next step is to create a 'mapping' function, the PL/SQL binding to Java. This will simply be:

```
dirlist@DEV816> create or replace
  2  procedure get_dir_list( p_directory in varchar2 )
  3  as language java
  4  name 'DirList.getList( java.lang.String )';
  5  /
Procedure created.
```

Now we are ready to go:

```
dirlist@DEV816> exec get_dir_list( '\tmp' );
PL/SQL procedure successfully completed.
```

```
dirlist@DEV816> select * from dir_list where rownum < 5;

FILENAME
------------------
lost+found
.rpc_door
ps_data
.pcmcia
```

and that's it. Now we can list the contents of a directory into this temporary table. We can then apply filters to this easily using LIKE and sort the output if we like as well.

Summary

UTL_FILE is an excellent utility you will most likely find use for in many of your applications. In this section, we covered the setup necessary for using UTL_FILE and described how it works. We've looked at some of the most common issues I see people running into with UTL_FILE such as accessing network drives in a Windows environment, hitting the 1023 byte limit, and handling exceptions. For each we presented the solutions. We also explored some utilities you might develop with UTL_FILE such as the UNLOADER described in Chapter 9 on *Data Loading*, the ability to read a directory presented here, or dumping a web page to disk as described above.