CHAPTER 28

External Procedures

In the early days of PL/SQL it was common to hear the question, "Can I call *whatever* from within Oracle?" Typically, *whatever* had something to do with sending email, running operating system commands, or using some non-PL/SQL feature. Although email has pretty much been a nonissue since Oracle began shipping the built-in UTL_SMTP and UTL_MAIL packages, there are by now quite a handful of alternatives to calling *whatever*. Here are the most common approaches:

- Write the program as a Java stored procedure and call the Java from PL/SQL.
- Use a database table or queue as a place to store the requests, and create a separate process to read and respond to those requests.
- Expose the program as a web service.
- Use a database pipe and write a daemon that responds to requests on the pipe.
- Write the program in C and call it as an external procedure.

Java may work well, and it can be fast enough for many applications. Queuing is a very interesting technology, but even if you are simply using plain tables, this approach requires two Oracle database sessions: one to write to the queue and one to read from it. Moreover, two sessions means two different transaction spaces, and that might be a problem for your application. Database pipe-based approaches also have the two-session problem, not to mention the challenge of packing and unpacking the contents of the pipe. In addition, handling many simultaneous requests using any of these approaches might require you to create your own listener and process-dispatching system.

Those are all reasons to consider the final option. *External procedures* allow PL/SQL to do almost anything that any other language can do, and can remedy the shortcomings of the other approaches just mentioned. But just how do external procedures work? Are they secure? How can you build your own? What are their advantages and disadvantages? This chapter addresses these questions and provides examples of commonly used features of external procedures.

By the way, examples in this chapter make use of the GNU compiler collection (GCC), which, like Oracle, runs practically everywhere.[1] I love GCC, but it won't work in every situation; you might need to use the compiler that is "native" on your hardware.

Introduction to External Procedures

To call an external program from inside Oracle, the program must run as a shared library. You probably know this type of program as a DLL (dynamically linked library) file on Microsoft operating systems; on Solaris, AIX, and Linux, you'll usually see shared libraries with a *.so* (shared object) file extension, or *.sl* (shared library) on HP-UX. In theory, you can write the external routine in any language you wish, but your compiler and linker will need to generate the appropriate shared library format that is callable from C. You "publish" the external program by writing a special PL/SQL wrapper, known as a *call specification*. If the external function returns a value, it maps to a PL/SQL function; if the external function returns nothing, it maps to a PL/SQL procedure.

Example: Invoking an Operating System Command

Our first example allows a PL/SQL program to execute any operating system–level command. Eh? I hope your mental security buzzer is going off—that sounds like a really dangerous thing to do, doesn't it? Despite several security hoops you have to jump through to make it work, your database administrator will still object to granting wide permissions to run this code. Just try to suspend your disbelief as we walk through the examples.

The first example consists of a very simple C function, extprocsh, which accepts a string and passes it to the system function for execution:

```
int extprocshell(char *cmd)
{
   return system(cmd);
}
```

The function returns the result code as provided by system, a function normally found in the C runtime library (*libc*) on Unix, or in *msvcrt.dll* on Microsoft platforms.

After saving the source code in a file named *extprocsh.c*, I can use the GNU C compiler to generate a shared library. On my 64-bit Solaris machine running GCC 3.4.2 and

1. GCC is free to use; there are at least two versions for Microsoft Windows. This chapter uses the one from MinGW.

Oracle Database 10*g* Release 2, I used the following compiler command (note that GCC options vary on different Unix/Linux distributions):

```
gcc -m64 extprocsh.c -fPIC -G -o extprocsh.so
```

Similarly, on Microsoft Windows XP Pro running GCC 3.2.3 from Minimal GNU for Windows (MinGW), also with Oracle Database 10*g* Release 2, this works:

```
c:\MinGW\bin\gcc extprocsh.c -shared -o extprocsh.dll
```

These commands generate a shared library file, *extprocsh.so* or *extprocsh.dll*. Now I need to put the library file somewhere that Oracle can find it. Depending on your Oracle version, that may be easier said than done! Table 28-1 gives you a clue as to where to put the files.

Table 28-1. Location of shared library files

Version	Default "allowed" location	Means of specifying nondefault location
Oracle8 Database, Oracle8*i* Database, Oracle9*i* Database Release 1	Anywhere readable by the Oracle process	Not applicable
Oracle9*i* Release 2 and later	*$ORACLE_HOME/lib* and/or *$ORACLE_HOME/bin* (varies by Oracle version and platform; *lib* is typical for Unix and *bin* for Microsoft Windows)	Edit listener configuration file and supply path value(s) for ENVS="EXTPROC_DLLS..." property (see the section "Oracle Net Configuration" on page 1248)

After copying the file and/or making adjustments to the listener, I also need to define a "library" inside Oracle to point to the DLL:

```
CREATE OR REPLACE LIBRARY extprocshell_lib
   AS '/u01/app/oracle/local/lib/extprocsh.so';   -- Unix/Linux

CREATE OR REPLACE LIBRARY extprocshell_lib
   AS 'c:\oracle\local\lib\extprocsh.dll';      -- Microsoft
```

Don't by confused by the term *library* here; it's really just a filename alias that can be used in Oracle's namespace. Also note that performing this step requires Oracle's CREATE LIBRARY privilege, which is one of the security hoops I mentioned earlier.

Now I can create a PL/SQL call specification that uses the newly created library:

```
FUNCTION shell(cmd IN VARCHAR2)
   RETURN PLS_INTEGER
AS
   LANGUAGE C
   LIBRARY extprocshell_lib
   NAME "extprocshell"
   PARAMETERS (cmd STRING, RETURN INT);
```

That's all there is to it! Assuming that the DBA has set up the system environment to support external procedures (see the section "Specifying the Listener Configuration" on page 1248), shell is now usable anywhere you can invoke a PL/SQL function—SQL*Plus, Perl, Pro*C, and so on. From an application programming perspective, calling an external procedure is indistinguishable from calling a conventional procedure. For example:

```
DECLARE
   result PLS_INTEGER;
BEGIN
   result := shell('cmd');
END;
```

Or even:

```
SQL> SELECT shell('cmd') FROM DUAL;
```

If successful, this will return zero:

```
SHELL('cmd')
-------------
          0
```

Keep in mind that if the operating system command normally displays output to stdout or stderr, that output will go to the bit bucket unless you modify your program to return it to PL/SQL. You can, subject to OS-level permissions, redirect that output to a file; here is a trivial example of saving a file containing a directory listing:

```
result := shell('ls / > /tmp/extproc.out'));                -- Unix/Linux
result := shell('cmd /c "dir c:\ > c:\temp\extproc.out"'));  -- Microsoft
```

These operating system commands will execute with the same privileges as the Oracle Net listener that spawns the *extproc* process. Hmmm, I'll bet your DBA or security guy will want to change *that*. Read on if you want to help.

Architecture of External Procedures

What happens under the covers when you invoke an external procedure? Let's first consider a case such as the example illustrated in the previous section, which uses the default external procedure "agent."

When the PL/SQL runtime engine learns from the compiled code that the program has been implemented externally, it looks for a TNS service named EXTPROC_CONNECTION_DATA, which must be known to the server via some Oracle Net naming method such as the *tnsnames.ora* file. As shown in Figure 28-1, the Oracle Net listener responds to the request by spawning a session-specific process called *extproc*, to which it passes the path to the DLL file along with the function name and any arguments. It is *extproc* that dynamically loads your shared library, sends needed arguments, receives its output, and transmits these results back to the caller. In this arrangement, only one *extproc* process runs for a given Oracle session; it launches with the first external procedure call and terminates when the session disconnects. For each distinct external procedure you call, this *extproc* process loads the associated shared library (if it hasn't already been loaded).

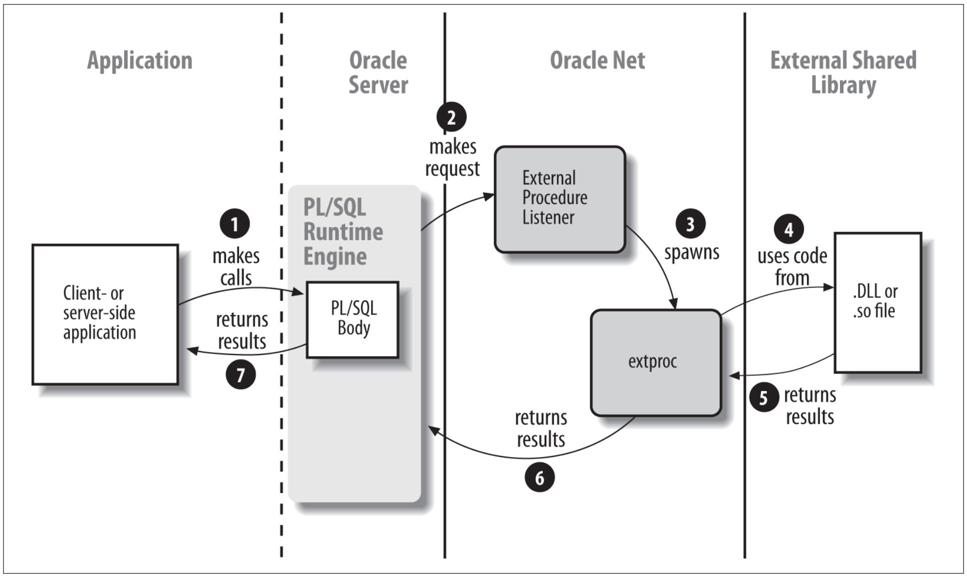

Figure 28-1. Invoking an external procedure that uses the default agent

Oracle has provided a number of features to help make external procedures usable and efficient:

Shared DLL
: The external C program must be in a shared dynamically linked library rather than in a statically linked module. Although deferring linking until runtime incurs some overhead, there should be memory savings when more than one session uses a shared library; the operating system allows some of the memory pages of the library to be shared by more than one process. Another benefit of using dynamically linked modules is that they can be created and updated more easily than statically linked programs. In addition, there can be many subprograms in a shared library (hence the term *library*). This mitigates the performance overhead by allowing you to load fewer files dynamically.

Separate memory space
: Oracle external procedures run in a separate memory space from the main database kernel processes. If the external procedure crashes, it won't step on kernel memory; the *extproc* process simply returns an error to the PL/SQL engine, which in turn reports it to the application. Writing an external procedure to crash the Oracle

server is possible, but it's no easier than doing so from a nonexternal procedure program.

Full transaction support
External procedures provide full transaction support; that is, they can participate fully in the current transaction. By accepting "context" information from PL/SQL, the procedure can call back to the database to fetch data, make SQL or PL/SQL calls, and raise exceptions. Using these features requires some low-level Oracle Call Interface (OCI) programming... but at least it's possible!

Multithreading (Oracle Database 10g and later)
Up through Oracle9*i* Database, each Oracle session that called an external procedure required a companion *extproc* process. For large numbers of users, the added overhead could be significant. Starting with Oracle Database 10*g*, though, the DBA can configure a multithreaded "agent" that services each request in a thread rather than a separate process. You will need to ensure that your C program is thread-safe if you go this route. See the section "Setting Up Multithreaded Mode" on page 1252 for more information about using this feature.

Despite their many features and benefits, external procedures are not a perfect match for every application: Oracle's architecture requires an unavoidable amount of interprocess communication. This is the tradeoff required for the safety of separating the external procedure's memory space from that of the database server.

Oracle Net Configuration

Let's take a look at how you would set up a simple configuration that will support external procedures while closing up some of the glaring security gaps.

Specifying the Listener Configuration

It is the Oracle Net communications layer that provides the conduit between PL/SQL and the shared libraries. Although default installations of Oracle8*i* Database and later generally provide some support for external procedures, you probably won't want to use the out-of-the-box configuration until Oracle has made some significant security enhancements.

At the time we were writing the third edition of this book, Oracle was suffering a bit of a black eye from a security vulnerability arising from the external procedures feature. Specifically, a remote attacker could connect via the Oracle Net TCP/IP port (usually 1521) and run *extproc* with no authentication. Although Oracle closed up that particular vulnerability, the conventional wisdom of securing Oracle includes that shown in the following note.

Keep Oracle listeners behind a firewall; never expose a listener port to the Internet or to any other untrusted network.

Getting the listener set up properly involves modifying the *tnsnames.ora* file and the *listener.ora* file (either by hand or by using the Oracle Net Manager frontend). Here, for example, is a simple *listener.ora* file that sets up an external procedure listener that is separate from the database listener:

```
### regular listener (to connect to the database)

LISTENER =
   (ADDRESS = (PROTOCOL = TCP)(HOST = hostname)(PORT = 1521))

SID_LIST_LISTENER =
   (SID_DESC =
      (GLOBAL_DBNAME = global_name)
      (ORACLE_HOME = oracle_home_directory)
      (SID_NAME = SID)
   )

### external procedure listener

EXTPROC_LISTENER =
   (ADDRESS = (PROTOCOL = IPC)(KEY = extprocKey))

SID_LIST_EXTPROC_LISTENER =
   (SID_DESC =
      (SID_NAME = extprocSID)
      (ORACLE_HOME = oracle_home_directory)
      (ENVS="EXTPROC_DLLS=shared_object_file_list,other_envt_vars")
      (PROGRAM = extproc)
   )
```

where:

hostname, global_name
: The name or IP address of this machine and the fully qualified name of the database, respectively. In the example, these parameters apply to the database listener only, not to the external procedure listener.

extprocKey
: A short identifier used by Oracle Net to distinguish this listener from other potential interprocess communication (IPC) listeners. Its actual name is arbitrary because your programs will never see it. Oracle uses EXTPROC0 or EXTPROC1 as the default name for the first Oracle Net installation on a given machine. This identifier must be the same in the address list of *listener.ora* and in the *tnsnames.ora* file.

oracle_home_directory

The full pathname to your *ORACLE_HOME* directory, such as */u01/app/oracle/oracle/product/10.2.0/db_1* on Unix or *C:\oracle\product\10.2.0\db_1* on Microsoft Windows. Notice that there are no quotation marks around the directory name and no trailing slash.

extprocSID

An arbitrary unique identifier for the external procedure listener. In the default installation, Oracle uses the value PLSExtProc.

ENVS="EXTPROC_DLLS=shared_object_file_list"

Available in Oracle9*i* Database Release 2 and later, the ENVS clause sets up environment variables for the listener. The list is colon-delimited, and each element must be the fully qualified pathname to a shared object file.

There are some special keywords you can use in the list. For no security at all, you can use the ANY keyword, which lets you use any shared library that is visible to the operating system user running the external procedure listener. In other words:

```
ENVS="EXTPROC_DLLS=ANY"
```

For maximum security, use the ONLY keyword to limit execution of those shared libraries given by the colon-delimited list. Here is an example from my Solaris machine that shows what this might look like:

```
(ENVS="EXTPROC_DLLS=ONLY:/u01/app/oracle/local/lib/extprocsh.so:/u01/app/
oracle/local/lib/RawdataToPrinter.so")
```

And here is an entry from my laptop machine, which runs a Microsoft Windows operating system:

```
(ENVS="EXTPROC_DLLS=ONLY:c:\oracle\admin\local\lib\extprocsh.dll:c:\oracle\
admin\local\lib\RawDataToPrinter.dll")
```

Here, the colon symbol has two different meanings: as the list delimiter or as the drive letter separator. Also note that although I've shown only two library files, you can include as many as you like.

If you omit the ANY and ONLY keywords but still provide a list of files, both the default directories and the explicitly named files are available.

other_envt_vars

You can set values for environment variables needed by shared libraries by adding them to the ENVS setting of the external procedure listener. A commonly needed value on Unix is LD_LIBRARY_PATH:

```
(ENVS="EXTPROC_DLLS=shared_object_file_list,LD_LIBRARY_PATH=/usr/local/lib")
```

Use commas to separate the list of files and each environment variable.

Security Characteristics of the Configuration

The configuration established here accomplishes two important security objectives:

- It allows the system administrator to run the external procedure listener as a user account with limited privileges. By default, the listener would run as the account that runs the Oracle server.
- It limits the external procedure listener to accept only IPC connections from the local machine, as opposed to TCP/IP connections from anywhere.

But we're not quite done. The *tnsnames.ora* file for the database in which the callout originates will need an entry like the following:

```
EXTPROC_CONNECTION_DATA =
   (DESCRIPTION =
      (ADDRESS = (PROTOCOL = IPC)(KEY = extprocKey))
      (CONNECT_DATA = (SID = extprocSID) (PRESENTATION = RO))
   )
```

You'll recognize most of these settings from the earlier listener configuration. Note that the values you used in the listener for *extprocKey* and *extprocSID* must match their respective values here. The optional PRESENTATION setting is intended to improve performance a bit; it tells the server, which might be listening for different protocols, to assume that the client wants to communicate using the protocol known as Remote-Ops (hence the RO).

You'll want to be careful about what privileges the supplemental listener account has, especially regarding its rights to modify files owned by the operating system or by the oracle account. Also, by setting the TNS_ADMIN environment variable on Unix (or in the Windows Registry), you can relocate the external procedure listener's *listener.ora* and *sqlnet.ora* files to a separate directory. This may be another aspect of an overall approach to security.

Setting up these configuration files and creating supplemental OS-level user accounts may seem rather distant from day-to-day PL/SQL programming, but these days, security is everybody's business!

Oracle professionals should keep up with Oracle's security alerts page. The external procedures problem I mentioned back in the section "Specifying the Listener Configuration" on page 1248 first appeared as alert number 29, but every Oracle shop should regularly review the entire list of issues to discover what workarounds or patches to employ.

Setting Up Multithreaded Mode

Oracle Database 10*g* introduced a way for multiple sessions to share a single external procedure process. Although this feature takes a bit of effort to set up, it could pay off when you have many users running external procedures. Here are the minimum steps required for your DBA to turn on multithreaded mode:

1. Shut down the external procedure listener. If you have configured a separate listener for it as recommended, this step is simply:

   ```
   OS> lsnrctl stop extproc_listener
   ```

2. Edit *listener.ora*: first, change your *extprocKey* (which by default would be EXTPROC0 or EXTPROC1) to PNPKEY; second, to eliminate the possibility of any dedicated listeners, delete the entire SID_LIST_EXTPROC_LISTENER section.
3. Edit *tnsnames.ora*, changing your *extprocKey* to PNPKEY.
4. Restart the external procedure listener; for example:

   ```
   OS> lsnrctl start extproc_listener
   ```

5. At the operating system command prompt, be sure you have set a value for the AGTCTL_ADMIN environment variable. The value should consist of a fully qualified directory path; this tells *agtctl* where to store its settings. (If you don't set AGTCTL_ADMIN, but do have TNS_ADMIN set, the latter will be used instead.)
6. If you need to send any environment variables to the agent, such as EXTPROC_DLLS or LD_LIBRARY_PATH, set these in the current operating system session. Here are some examples (if you're using the *bash* shell or equivalent):

   ```
   OS> export EXTPROC_DLLS=ANY
   OS> export LD_LIBRARY_PATH=/lib:/usr/local/lib/sparcv9
   ```

7. Assuming that you are still using the external procedure's default listener SID (that is, PLSExtProc), run the following:

   ```
   OS> agtctl startup extproc PLSExtProc
   ```

To see if it's working, you can use the lsnrctl services command:

```
OS> lsnrctl services extproc_listener
...
Connecting to (ADDRESS=(PROTOCOL=IPC)(KEY=PNPKEY))
Services Summary...
Service "PLSExtProc" has 1 instance(s).
  Instance "PLSExtProc", status READY, has 1 handler(s) for this service...
    Handler(s):
      "ORACLE SERVER" established:0 refused:0 current:0 max:5 state:ready
         PLSExtProc
         (ADDRESS=(PROTOCOL=ipc)(KEY=#5746.1.4))
```

This output is what we hoped to see; the agent is listed in state "ready" and is not labeled as dedicated. This command also shows stats on the number of sessions; in the preceding output, everything is 0 except the maximum number of sessions, which defaults to 5.

Internally, a multithreaded agent uses its own listener/dispatcher/worker bee arrangement, allowing each session request to get handed off to its own thread of execution. You can control the numbers of tasks using "agtctl set" commands. For example, to modify the maximum number of sessions, first shut down the agent:

```
OS> agtctl shutdown PLSExtProc
```

Then set max_sessions:

```
OS> agtctl set max_sessions n PLSExtProc
```

where *n* is the maximum number of Oracle sessions that can connect to the agent simultaneously.

Finally, restart the agent:

```
OS> agtctl startup extproc PLSExtProc
```

When tuning your setup, there are several parameter settings to be aware of, listed in the following table.

Parameter	Description	Default
max_dispatchers	Maximum number of dispatcher threads, which hand off requests to the task threads	1
max_task_threads	Maximum number of "worker bee" threads	2
max_sessions	Maximum number of Oracle sessions that can be serviced by the multithreaded extproc process	5
listener_address	Addresses with which the multithreaded process "registers" with an already running listener	`(ADDRESS_LIST= (ADDRESS= (PROTOCOL=IPC) (KEY=PNPKEY)) (ADDRESS= (PROTOCOL=IPC) (KEY=`*`listenerSID`*`)) (ADDRESS= (PROTOCOL=TCP) (HOST=127.0.0.1) (PORT=1521)))`

By the way, while testing this feature, I discovered that whenever I bounced the listener, afterward I needed to bounce the agent as well. Fortunately, the *agtctl* utility kindly remembers any parameter adjustments you have made from the default values.

Some experimentation may be needed to optimize the agent-specific parameters against the number of agent processes. While I have not experimented enough with multithreaded agents to offer any rules of thumb, let's at least take a look at the changes required to use two multithreaded agents. Follow the steps given earlier, but this time, in step 7, you will start two agents with unique names:

```
OS> agtctl startup extproc PLSExtProc_001
...
OS> agtctl startup extproc PLSExtProc_002
```

You must also modify *tnsnames.ora* (step 3) to be aware of these new agent names. Because you probably want Oracle Net to load balance across the agents, edit the EXTPROC_CONNECTION_DATA section of *tnsnames.ora* to be:

```
EXTPROC_CONNECTION_DATA =
   (DESCRIPTION_LIST =
      (LOAD_BALANCE = TRUE)
      (DESCRIPTION =
         (ADDRESS = (PROTOCOL = IPC)(KEY = PNPKEY))
         (CONNECT_DATA = (SID = PLSExtProc_001)(PRESENTATION = RO))
      )
      (DESCRIPTION =
         (ADDRESS = (PROTOCOL = ipc)(key = PNPKEY))
         (CONNECT_DATA = (SID = PLSExtProc_002)(PRESENTATION = RO))
      )
   )
```

You need to add the DESCRIPTION_LIST parameter and include one description section for each agent.

With this new configuration, each time Oracle Net receives a request from PL/SQL to connect to an external procedure, it will randomly connect to one of the agents listed; if the connection fails (for example, because it already has the maximum number of sessions connected), Oracle Net will try the other agent, until it either makes a successful connection or fails to connect to any of them. Such a failure will result in the error *ORA-28575: unable to open RPC connection to external procedure agent*.

You can read more about multithreaded mode and *agtctl* in Oracle's *Application Development Guide—Fundamentals* and the *Heterogeneous Connectivity Administrator's Guide*. Oracle Net's load balancing features are described in the *Net Services Administrator's Guide* and the *Net Services Reference*.

One final point: when using multithreaded agents, the C program that implements an external procedure must be *thread-safe*, and writing such a beasty is not necessarily trivial. See the notes at the end of this chapter for a few of the caveats.

Creating an Oracle Library

The SQL statement CREATE LIBRARY defines an alias in the Oracle data dictionary for the external shared library file, allowing the PL/SQL runtime engine to find the library when it is called. The only users who can create libraries are administrators and those to whom they have granted the CREATE LIBRARY or CREATE ANY LIBRARY privilege.

The general syntax for the CREATE LIBRARY command is:

```
CREATE [ OR REPLACE ] LIBRARY library_name
AS
   'path_to_file' [ AGENT 'agent_db_link' ] ;
```

where:

library_name
: Is a legal PL/SQL identifier. This name will be used in subsequent bodies of external procedures that need to call the shared object (or DLL) file. The library name cannot be the same as a table, a top-level PL/SQL object, or anything else in the main namespace.

path_to_file
: Specifies the fully qualified pathname to the shared object (or DLL) file, enclosed in single quotes.

  In Oracle9*i* Database, it became possible to use environment variables in *path_to_file*. In particular, if the operating system–level account sets the variable before starting the listener, you can put this variable in the CREATE LIBRARY statement; for example:

```
CREATE LIBRARY extprocshell_lib AS '${ORACLE_HOME}/lib/extprocsh.so'; -- Unix
CREATE LIBRARY extprocshell_lib AS '%{ORACLE_HOME}%\bin\extprocsh.dll'; -- MS
```

  This may be a good thing to do for the sake of script portability.

  You can also use an environment variable that you supply via EXTPROC_DLLS in the *listener.ora* file, as discussed earlier.

AGENT 'agent_db_link'
: (Optional) Is a database link to which the library owner has access. You must make sure that there is an entry in *tnsnames.ora* for the service name you specify when creating *agent_db_link*, and that the entry includes an external procedure address and connection data. Using the AGENT clause allows the external procedure to run on a different database server, although it must still be on the same machine. The AGENT clause was introduced in Oracle9*i* Database.

Here are some things to keep in mind when issuing a CREATE LIBRARY statement:

- The statement must be executed by the DBA or by a user who has been granted CREATE LIBRARY or CREATE ANY LIBRARY privileges.
- As with most other database objects, libraries are owned by a specific Oracle user (schema). The owner automatically has execution privileges, and can grant and revoke the EXECUTE privilege on the library to and from other users.

- Other users who have received the EXECUTE privilege on a library can refer to it in their own call specs using *owner.library* syntax, or they can create and use synonyms for the library if desired.
- Oracle doesn't check whether the named shared library file exists when you execute the CREATE LIBRARY statement. Nor will it check when you later create an external procedure declaration for a function in that library. If you have an error in the path, you won't know it until the first time you try to execute the function.

You need to create only a single Oracle library in this fashion for each shared library file you use. There can be any number of callable C functions in the library file and any number of call specifications that refer to the library.

Let's take a closer look at how to write a PL/SQL subprogram that maps the desired routine from the shared library into a PL/SQL-callable form.

Writing the Call Specification

An external procedure can serve as the implementation of any program unit other than an anonymous block. In other words, a call specification can appear in a top-level procedure or function, a packaged procedure or function, or an object method. What's more, you can define the call spec in either the specification or the body of packaged program units (or in either the spec or body of object types). Here are some schematic examples:

```
CREATE FUNCTION name (args) RETURN datatype
AS callspec;
```

You should recognize the form shown here as that of the shell function shown earlier in the chapter. You can also create a procedure:

```
CREATE PROCEDURE name
AS callspec;
```

In this case, the corresponding C function would be typed void.

The next form shows a packaged function that does not need a package body:

```
CREATE PACKAGE pkgname
AS
   FUNCTION name RETURN datatype
   AS callspec;
END;
```

However, when the time comes to modify the package, you will have to recompile the specification. Depending on the change you need to make, you may considerably reduce the recompilation ripple effect by moving the call spec into the package body:

```
CREATE PACKAGE pkgname
AS
```

```
   PROCEDURE name;
END;

CREATE PACKAGE BODY pkgname
AS
   PROCEDURE name
   AS callspec;
END;
```

Unpublished or private program units inside packages can also be implemented as external procedures. And finally, using a call spec in an object type method is quite similar to using it in a package; that is, you can put the call spec in the object type specification or in the corresponding type body.

The Call Spec: Overall Syntax

It is the AS LANGUAGE clause[2] that distinguishes the call spec from a regular stored program.

Syntactically, the clause looks like this:

```
AS LANGUAGE C
   LIBRARY library_name
   [ NAME external_function_name ]
   [ WITH CONTEXT ]
   [ AGENT IN (formal_parameter_name) ]
   [ PARAMETERS (external_parameter_map) ] ;
```

where:

AS LANGUAGE C
: Another option here is AS LANGUAGE JAVA, as covered in Chapter 27. There are no other supported languages.

library_name
: Is the name of the library, as defined in a CREATE LIBRARY statement, which you have privilege to execute, either by owning it or by receiving the privilege.

external_function_name
: Is the name of the function as defined in the C language library. If the name is lowercase or mixed case, you must put double quotes around it. You can omit this parameter, in which case the name of the external routine must match your PL/SQL module's name (defaults to uppercase).

2. Oracle8 Database did not have this clause, offering instead a now-deprecated form, AS EXTERNAL.

WITH CONTEXT
: Indicates that you want PL/SQL to pass a "context pointer" to the called program. The called program must be expecting the pointer as a parameter of type OCIExtProcContext * (defined in the C header file *ociextp.h*).

  This "context" that you are passing via a pointer is an opaque data structure that contains Oracle session information. The called procedure doesn't need to manipulate the data structure's content directly; instead, the structure simply facilitates other OCI calls that perform various Oracle-specific tasks. These tasks include raising predefined or user-defined exceptions, allocating session-only memory (which gets released as soon as control returns to PL/SQL), and obtaining information about the Oracle user's environment.

AGENT IN (formal_parameter_name)
: Is a way of designating a different agent process, similar to the AGENT clause, on the library, but deferring the selection of the agent until runtime. The idea is that you pass in the value of the agent as a formal PL/SQL parameter to the call spec; it will supersede the name of the agent given in the library, if any. To learn a little more about the AGENT IN clause, see the section "Nondefault Agents" on page 1269.

PARAMETERS (external_parameter_map)
: Gives the position and datatypes of parameters exchanged between PL/SQL and C. The *external_parameter_map* is a comma-delimited list of elements that match positionally with the parameters in the C function or that supply additional properties.

Getting the mapping right is potentially the most complex task you face, so the next section spends a bit of time examining the wilderness of details.

Parameter Mapping: The Example Revisited

Consider for a moment the problems of exchanging data between PL/SQL and C. PL/SQL has its own set of datatypes that are only somewhat similar to those you find in C. PL/SQL variables can be NULL and are subject to three-valued truth table logic; C variables have no equivalent concept. Your C library might not know which national language character set you're using to express alphanumeric values. And should your C functions expect a given argument by value or by reference (pointer)?

I'd like to start with an example that builds on the shell program illustrated earlier in the chapter. When we last saw the shell function, it had no protection from being called with a NULL argument instead of a real command. It turns out that calling shell (NULL) results in the runtime error *ORA-01405: fetched column value is NULL*. That may be a perfectly acceptable behavior in some applications, but what if I prefer that the external procedure simply respond to a null input with a null output?

Properly detecting an Oracle NULL in C requires PL/SQL to transmit an additional parameter known as an *indicator variable*. Likewise, for the C program to return an Oracle NULL, it must return a separate indicator parameter back to PL/SQL. While Oracle sets and interprets this value automatically on the PL/SQL side, the C application will need to get and set this value explicitly.

It's probably simplest to illustrate this situation by looking at how the PL/SQL call spec will change:

```
FUNCTION shell(cmd IN VARCHAR2)
   RETURN PLS_INTEGER
AS
   LANGUAGE C
   LIBRARY extprocshell_lib
   NAME "extprocsh"
   PARAMETERS (cmd STRING, cmd INDICATOR, RETURN INDICATOR, RETURN INT);
```

Although the PL/SQL function's formal parameters can appear anywhere in the PARAMETERS mapping, the items in the mapping must correspond in position and in associated datatype with the parameters in the C function. Any RETURN mapping that you need to provide must be the last item on the list.

You can omit RETURN from the parameter map if you want Oracle to use the default mapping (explained later). This would actually be OK in this case, although the indicator still has to be there:

```
FUNCTION shell(cmd IN VARCHAR2)
   RETURN PLS_INTEGER
AS
   LANGUAGE C
   LIBRARY extprocshell_lib
   NAME "extprocsh"
   PARAMETERS (cmd STRING, cmd INDICATOR, RETURN INDICATOR);
```

The good news is that even though you've made a number of changes to the call spec compared with the version earlier in the chapter, a program that invokes the shell function sees no change in the number or datatypes of its parameters.

Let's turn now to the new version of the C program, which adds two parameters, one for each indicator:

```
#include <ociextp.h>

int extprocsh(char *cmd, short cmdInd, short *retInd)
{
   if (cmdInd == OCI_IND_NOTNULL)
   {
      *retInd = (short)OCI_IND_NOTNULL;
      return system(cmd);
   } else
   {
```

```
11          *retInd = (short)OCI_IND_NULL;
12          return 0;
13       }
14    }
```

The changes you'll notice are summarized in the following table.

Line(s)	Changes
1	This include file appears in the *%ORACLE_HOME%\oci\include* subdirectory on Microsoft platforms; on Unix-like machines, I've spotted this file in *$ORACLE_HOME/rdbms/demo* (Oracle9*i* Database) and *$ORACLE_HOME/rdbms/public* (Oracle Database 10*g*), although it may be somewhere else on your system.
3	Notice that the command indicator is short, but the return indicator is short *. That follows the argument-passing convention of using call by value for input parameters sent from PL/SQL to C, but call by reference for output or return parameters sent from C to PL/SQL.
5, 7	The indicator variable is either OCI_IND_NULL or OCI_IND_NOTNULL; these are special #define values from Oracle's include file. Here, explicit assignments in the code set the return indicator to be one or the other.
11–12	The return indicator takes precedence over the return of 0; the latter is simply ignored.

Here are some simple commands that will compile and link the preceding program on my 64-bit Solaris machine (still using GCC):

```
gcc -m64 extprocsh.c -fPIC -G -I$ORACLE_HOME/rdbms/public -o extprocsh.so
```

And here are the equivalent commands on my Microsoft Windows machine (this should all be entered on one line):

```
c:\MinGW\bin\gcc -Ic:\oracle\product\10.2.0\db_1\oci\include extprocsh.c
-shared -o extprocsh.dll
```

And now, steel yourself to face the intimidating details of parameter mapping.

Parameter Mapping: The Full Story

As shown in the previous section, when you are moving data between PL/SQL and C, each PL/SQL datatype maps to an external datatype, identified by a PL/SQL keyword, which in turn maps to an allowed set of C types.

You identify an external datatype in the PARAMETERS clause with a keyword known to PL/SQL. In some cases, the external datatypes have the same names as the C types, but in others they don't. For example, if you pass a PL/SQL variable of type PLS_INTEGER, the corresponding default external type is INT, which maps to an int in C. But Oracle's VARCHAR2 type uses the STRING external datatype, which normally maps to a char * in C.

Table 28-2 lists all the possible datatype conversions supported by Oracle's PL/SQL-to-C interface. Note that the allowable conversions depend on both the datatype and the mode of the PL/SQL formal parameter, as the previous example illustrated. The defaults, if ambiguous, are shown in bold in the table.

Table 28-2. Legal mappings of PL/SQL and C datatypes

		C datatypes for PL/SQL parameters that are:	
Datatype of PL/SQL parameter	**PL/SQL keyword identifying external type**	**N or function return values**	**IN OUT, OUT, or any parameter designated as being passed BY REFERENCE**
Long integer family: BINARY_INTEGER, BOOLEAN, PLS_INTEGER	**INT**, UNSIGNED INT, CHAR, UNSIGNED CHAR, SHORT, UNSIGNED SHORT, LONG, UNSIGNED LONG, SB1, UB1, SB2, UB2, SB4, UB4, SIZE_T	int, unsigned int, char, unsigned char, short, unsigned short, long, unsigned long, sb1, ub1, sb2, ub2, sb4, ub4, size_t	Same list of types as at left, but use a pointer (for example, int * rather than int)
Short integer family: NATURAL, NATURALN, POSITIVE, POSITIVEN, SIGNTYPE	Same as above, except default is UNSIGNED INT	Same as above, except default is unsigned int	Same as above, except default is unsigned int *
Character family: VARCHAR2, CHAR, NCHAR, LONG, NVARCHAR2, VARCHAR, CHARACTER, ROWID	**STRING**, OCISTRING	**char ***, OCIString *	**char ***, OCIString *
NUMBER	OCINUMBER	OCINumber *	OCINumber *
DOUBLE PRECISION	DOUBLE	double	double *
FLOAT, REAL	FLOAT	float	float *
RAW, LONG RAW	**RAW**, OCIRAW	**unsigned char ***, OCIRaw *	**unsigned char ***, OCIRaw *
DATE	OCIDATE	OCIDate *	OCIDate *
Timestamp family: TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE	OCIDATETIME	OCIDateTime *	OCIDateTime *
INTERVAL DAY TO SECOND, INTERVAL YEAR TO MONTH	OCIINTERVAL	OCIInterval *	OCIInterval *
BFILE, BLOB, CLOB, NCLOB	OCILOBLOCATOR	OCILOBLOCATOR *	OCILOBLOCATOR **
Descriptor of user-defined type (collection or object)	TDO	OCIType *	OCIType *
Value of user-defined collection	OCICOLL	OCIColl **, OCIArray **, OCITable **	OCIColl **, OCIArray **, OCITable **
Value of user-defined object	DVOID	dvoid *	dvoid * (use dvoid ** for nonfinal types that are IN OUT or OUT)

In some simple cases where you are passing only numeric arguments and where the defaults are acceptable, you can actually omit the PARAMETERS clause entirely. However, you must use it when you want to pass indicators or other data properties.

Each piece of supplemental information we want to exchange will be passed as a separate parameter, and will appear both in the PARAMETERS clause and in the C language function specification.

More Syntax: The PARAMETERS Clause

The PARAMETERS clause provides a comma-delimited list that may contain five different kinds of elements:

- The name of the parameter followed by the external datatype identifier
- The keyword RETURN and its associated external datatype identifier
- A "property" of the PL/SQL parameter or return value, such as a nullness indicator or an integer corresponding to its length
- The keyword CONTEXT, which is a placeholder for the context pointer
- The keyword SELF, in the case of an external procedure for an object type member method

Elements (other than CONTEXT) follow the syntax pattern:

```
{pname | RETURN | SELF} [property] [BY REFERENCE] [external_datatype]
```

If your call spec includes WITH CONTEXT, the corresponding element in the parameter list is simply:

```
CONTEXT
```

By convention, if you have specified WITH CONTEXT, you should make CONTEXT the first argument because that is its default location if the rest of the parameter mappings are defaulted.

Parameter entries have the following meanings:

pname | RETURN | SELF
: The name of the parameter as specified in the formal parameter list of the PL/SQL module, or the keyword RETURN, or the keyword SELF (in the case of a member method in an object type). PL/SQL parameter names do not need to be the same as the names of formal parameters in the C language routine. However, parameters in the PL/SQL parameter list must match one for one, in order, those in the C language specification.

property
: One of the following: INDICATOR, INDICATOR STRUCT, LENGTH, MAXLEN, TDO, CHARSETID, or CHARSETFORM. These are described in the next section.

BY REFERENCE

Pass the parameter by reference. In other words, the module in the shared library is expecting a pointer to the parameter rather than its value. BY REFERENCE only has meaning for scalar IN parameters that are not strings, such as BINARY_INTEGER, PLS_INTEGER, FLOAT, DOUBLE PRECISION, and REAL. All others (IN OUT and OUT parameters, as well as IN parameters of type STRING) are *always* passed by reference, and the corresponding C prototype must specify a pointer.

external_datatype

The external datatype keyword from the second column of Table 28-2. If this is omitted, the external datatype will default as indicated in the table.

PARAMETERS Properties

This section describes each possible property you can specify in a PARAMETERS clause.

The INDICATOR property

The INDICATOR property is a flag to denote whether the parameter is null, and has the following characteristics:

Allowed external types

These include short (the default), int, and long.

Allowed PL/SQL types

All scalars can use an INDICATOR; to pass indicator variables for composite types such as user-defined objects and collections, use the INDICATOR STRUCT property.

Allowed PL/SQL modes

These include IN, IN OUT, OUT, and RETURN.

Call mode

By value for IN parameters (unless BY REFERENCE is specified), and by reference for IN OUT, OUT, and RETURN variables.

You can apply this property to any parameter, in any mode, including RETURNs. If you omit an indicator, PL/SQL is supposed to think that your external routine will always be non-null (but it's not that simple; see the sidebar "Indicating Without Indicators?" on page 1264).

When you send an IN variable and its associated indicator to the external procedure, Oracle sets the indicator's value automatically. However, if your C module is returning a value in a RETURN or OUT parameter and an indicator, your C code must set the indicator value.

For an IN parameter, the indicator parameter in your C function might be:

```
short pIndicatorFoo
```

Or for an IN OUT parameter, the indicator might be:

```
short *pIndicatorFoo
```

In the body of your C function, you should use the #define constants OCI_IND_NOTNULL and OCI_IND_NULL, which will be available in your program if you #include *oci.h*. Oracle defines these as:

```
typedef sb2 OCIInd;
#define OCI_IND_NOTNULL (OCIInd)0          /* not NULL */
#define OCI_IND_NULL (OCIInd)(-1)          /* NULL */
```

Indicating Without Indicators?

What happens if you don't specify an indicator variable for a string and then return an empty C string? I wrote a short test program to find out:

```
void mynull(char *outbuff){
    outbuff[0] = '\0';
}
```

The call spec could look like this:

```
CREATE OR REPLACE PROCEDURE mynull
   (res OUT VARCHAR2)
AS
   LANGUAGE C
   LIBRARY mynulllib
   NAME "mynull";
```

When invoked as an external procedure, PL/SQL does actually interpret this parameter value as a NULL. The reason appears to be that the STRING external type is special; you can also indicate a NULL value to Oracle by passing a string of length 2 where the first byte is \0. (This works only if you omit a LENGTH parameter.)

But you probably shouldn't take this lazy way out; use an indicator instead!

The LENGTH property

The LENGTH property is an integer indicating the number of characters in a character parameter, and has the following characteristics:

Allowed external types
: int (the default), short, unsigned short, unsigned int, long, unsigned long

Allowed PL/SQL types
: VARCHAR2, CHAR, RAW, LONG RAW

Allowed PL/SQL modes
: IN, IN OUT, OUT, RETURN

Call mode
: By value for IN parameters (unless BY REFERENCE is specified), and by reference for IN OUT, OUT, and RETURN variables

The LENGTH property is mandatory for RAW and LONG RAW, and is available as a convenience to your C program for the other datatypes in the character family. When passing a RAW from PL/SQL to C, Oracle will set the LENGTH property; however, your C program must set LENGTH if you need to pass the RAW data back.

For an IN parameter, the indicator parameter in your C function might be:

```
int pLenFoo
```

Or for an OUT or IN OUT parameter, it might be:

```
int *pLenFoo
```

The MAXLEN property

The MAXLEN property is an integer indicating the maximum number of characters in a string parameter, and has the following characteristics:

Allowed external types
: int (the default), short, unsigned short, unsigned int, long, unsigned long

Allowed PL/SQL types
: VARCHAR2, CHAR, RAW, LONG RAW

Allowed PL/SQL modes
: IN OUT, OUT, RETURN

Call mode
: By reference

MAXLEN applies to IN OUT, OUT, and RETURN parameters and to no other mode. If you attempt to use it for an IN, you'll get the compile-time error *PLS-00250: incorrect usage of MAXLEN in parameters clause*.

Unlike the LENGTH parameter, the MAXLEN data is always passed by reference.

Here's an example of the C formal parameter:

```
int *pMaxLenFoo
```

The CHARSETID and CHARSETFORM properties

The CHARSETID and CHARSETFORM properties are flags denoting information about the character set, and have the following characteristics:

Allowed external types
: unsigned int (the default), unsigned short, unsigned long

Allowed PL/SQL types
: VARCHAR2, CHAR, CLOB

Allowed PL/SQL modes
: IN, IN OUT, OUT, RETURN

Call mode
: By reference

If you are passing data to the external procedure that is expressed in a nondefault character set, these properties will let you communicate the character set's ID and form to the called C program. The values are read-only and should not be modified by the called program. Here is an example of a PARAMETERS clause that includes character set information:

```
PARAMETERS (CONTEXT, cmd STRING, cmd INDICATOR, cmd CHARSETID,
            cmd CHARSETFORM);
```

Oracle sets these additional values automatically, based on the character set in which you have expressed the cmd argument. For more information about Oracle's globalization support in the C program, refer to Oracle's OCI documentation.

Raising an Exception from the Called C Program

The shell program shown earlier in the chapter is very, um, "C-like": it is a function whose return value contains the status code, and the caller must check the return value to see if it succeeded. Wouldn't it make more sense—in PL/SQL, anyway—for the program to be a procedure that simply raises an exception when there's a problem? Let's take a brief look at how to perform the OCI equivalent of RAISE_APPLICATION_ERROR.

In addition to the easy change from a function to a procedure, there are several other things I need to do:

- Pass in the context area.
- Decide on an error message and an error number in the 20001–20999 range.
- Add a call to the OCI service routine that raises an exception.

The changes to the call spec are trivial:

```
/* File on web: extprocsh.sql */ PROCEDURE shell(cmd IN VARCHAR2)
AS
   LANGUAGE C
   LIBRARY extprocshell_lib
   NAME "extprocsh"
   WITH CONTEXT
   PARAMETERS (CONTEXT, cmd STRING, cmd INDICATOR);
/
```

(I also removed the return parameter and its indicator.) The following code shows how to receive and use the context pointer in the call needed to raise the exception:

```
       /* File on web: extprocsh.c */
1      #include <ociextp.h>
2      #include <errno.h>
3
4      void extprocsh(OCIExtProcContext *ctx, char *cmd, short cmdInd)
5      {
6         int excNum = 20001;  # a convenient number :->
7         char excMsg[512];
8         size_t excMsgLen;
9
10         if (cmdInd == OCI_IND_NULL)
11            return;
12
13         if (system(cmd) != 0)
14         {
15            sprintf(excMsg, "Error %i during system call: %.*s", errno, 475,
16                  strerror(errno));
17            excMsgLen = (size_t)strlen(excMsg);
18
19            if (OCIExtProcRaiseExcpWithMsg(ctx, excNum, (text *)excMsg, excMsgLen)
20                   != OCIEXTPROC_SUCCESS)
21               return;
22         }
23
24      }
```

Note the lines described in the following table.

Line(s)	Description
4	The first of the formal parameters is the context pointer.
6	You can use whatever number in Oracle's user-defined error number range you want; in general, I advise against hardcoding these values, but, er, this is a "do as I say, not as I do" example.
7	The maximum size for the text in a user-defined exception is 512 bytes.
8	A variable to hold the length of the error message text, which will be needed in the OCI call that raises the exception.
10–11	Here, I am translating the NULL argument semantics of the earlier function into a procedure: when called with NULL, nothing happens.
13	A zero return code from system means that everything executed perfectly; a nonzero code corresponds to either an error or a warning.
15, 17	These lines prepare the variables containing the error message and its length.
19–20	This OCI function, which actually raises the user-defined exception, is where the context pointer actually gets used.

Now, how do we compile this baby? First, on Unix/Linux:

```
gcc -m64 -extprocsh.c -I$ORACLE_HOME/rdbms/public -fPIC -shared -o extprocsh.so
```

That was easy enough. But on Microsoft Windows, I found that I needed an explicit *.def* file to define the desired entry point (more precisely, to exclude potential entry points found in Oracle's *oci.lib*):

```
/* File on web: build_extprocsh.bat */
echo LIBRARY extprocsh.dll > extprocsh.def
echo EXPORTS >> extprocsh.def
echo extprocsh >> extprocsh.def
```

Although we've had to break it to fit in the book's margins, the following line must be entered as one long string:

```
c:\MinGW\bin\gcc -c extprocsh.def extprocsh.c -IC:\oracle\product\10.2.0\
db_1\oci\
include C:\oracle\product\10.2.0\db_1\oci\lib\msvc\oci.lib-shared -o
extprocsh.dll
```

Here's what a test of the function *should* yield:

```
SQL> CALL shell('garbage');
CALL shell('garbage')
     *
ERROR at line 1:
ORA-20001: Error 2 during system call: No such file or directory
```

That is, you should get a user-defined −20001 exception with the corresponding text "no such file or directory." Unfortunately, I discovered that system does not always return meaningful error codes, and on some platforms the message is *ORA-20001: Error 0 during system call: Error 0*. (Fixing this probably requires using a call other than system —another reader exercise.)

A number of other OCI routines are unique to writing external procedures. Here is the complete list:

OCIExtProcAllocCallMemory
: Allocates memory that Oracle will automatically free when control returns to PL/SQL

OCIExtProcRaiseExcp
: Raises a predefined exception by its Oracle error number

OCIExtProcRaiseExcpWithMsg
: Raises a user-defined exception, including a custom error message (illustrated in the previous example)

OCIExtProcGetEnv
: Allows an external procedure to perform OCI callbacks to the database to execute SQL or PL/SQL

These all require the context pointer. Refer to Oracle's *Application Developer's Guide—Fundamentals* for detailed documentation and examples that use these routines.

Nondefault Agents

Starting with Oracle9*i* Database, it is possible to run external procedure agents via database links that connect to other local database servers. This functionality enables you to spread the load of running expensive external programs onto other database instances.

Even without other servers, running an external procedure through a nondefault agent launches a separate process. This can be handy if you have a recalcitrant external program. Launching it via a nondefault agent means that even if its *extproc* process crashes, it won't have any effect on other external procedures running in the session.

As a simple example of a nondefault agent, here is a configuration that allows an agent to run on the same database but in a separate extproc task. The *tnsnames.ora* file needs an additional entry such as:

```
agent0 =
  (DESCRIPTION =
    (ADDRESS = (PROTOCOL = IPC)(KEY=extprocKey))
    (CONNECT_DATA = (SID = PLSExtProc))
  )
```

Here, *extprocKey* can just be the same key as in your EXTPROC_CONNECTION_DATA entry.

Because agents are created with a database link, we'll need to create one of those:

```
SQL> CREATE DATABASE LINK agent0link
  2  CONNECT TO username IDENTIFIED BY password
  3  USING 'agent0';
```

Now, finally, the agent can appear in a CREATE LIBRARY statement such as:

```
CREATE OR REPLACE LIBRARY extprocshell_lib_with_agent
   AS 'c:\oracle\admin\local\lib\extprocsh.dll'
   AGENT 'agent0';
```

Any call spec that was written to use this library will authenticate and connect through this agent0 link, launching an extproc task separate from the default extproc task. In this way, you could separate tasks from each other (for example, you could send the thread-safe tasks to a multithreading agent and the others to dedicated agents).

Oracle also supports a dynamic arrangement that allows you to pass in the name of the agent as a parameter to the external procedure. To take advantage of this feature, use the AGENT IN clause in the call spec. For example (changes in boldface):

```
CREATE OR REPLACE PROCEDURE shell2 (name_of_agent IN VARCHAR2, cmd VARCHAR2)
AS
   LANGUAGE C
   LIBRARY extprocshell_lib
   NAME "extprocsh2"
   AGENT IN (name_of_agent)
   WITH CONTEXT
   PARAMETERS (CONTEXT, name_of_agent STRING, cmd STRING, cmd INDICATOR);
```

Notice that I had to include the name of the agent in the list of parameters. Oracle enforces a rule that every formal parameter must have a corresponding entry in the PARAMETERS clause, so I have to modify my external C library. In my case, I merely added a second entry point, extprocsh2, to the library with the following trivial function:

```
void extprocsh2(OCIExtProcContext *ctx, char *agent, char *cmd, short cmdInd)
{
   extprocsh(ctx, cmd, cmdInd);
}
```

My code just ignores the agent string. Now, though, I can invoke my shell2 procedure as in the following:

```
CALL shell2('agent0', 'whatever');
```

If you want your stored program to somehow invoke an external procedure on a remote machine, you have a couple of options. You could implement an external procedure on the local machine, which is just a "pass-through" program, making a C-based remote procedure call on behalf of PL/SQL. Alternatively, you could implement a stored PL/SQL program on the remote machine as an external procedure, and call it from the local machine via a database link. What isn't possible is setting up an external procedure listener to accept networked connections from a different machine.

A Debugging Odyssey

It turns out that some debuggers, including the GNU debugger (GDB), can attach to a running process and debug external procedures. Here is an outline of how I got this to work on Solaris and on Windows XP.

As a preliminary step on both platforms, I compiled the shared library file with the compiler option (*-g* in the case of GCC) needed to include symbolic information for the debugger. That was the easy part. During testing, I discovered the slap-hand-on-forehead fact that I could not debug my 64-bit Solaris external procedure with a 32-bit debugger, so I also had to build and install a 64-bit *gdb* executable. This step began in the root directory of the GDB source tree with the command:

```
OS> CC="gcc -m64" ./configure
```

At this point, the GDB build presented many surprises; it probably would have helped to have a competent system administrator looking over my shoulder!

Another preparatory step, although optional, involves running the script *dbgextp.sql*, which should be in the *plsql/demo* directory. If you're using Oracle Database 10*g* or later, you won't find this directory in your default $ORACLE_HOME distribution because Oracle moved the entire directory to the Companion CD. However, you may be able to extract the *plsql/demo* directory with a command like this (again, although it has been broken to fit the page margins, this should be entered as one long string):

```
OS> jar xvf /cdrom/stage/Components/oracle.rdbms.companion/10. x.x.x.x
    /1/DataFiles/filegroup1.jar
```

If you do manage to locate the *dbgextp.sql* file, you'll find that it contains some useful inline comments, which you should definitely read. Then run the script as yourself (not as SYS) to build a package named DEBUG_EXTPROC. This package contains a procedure whose sole purpose in life is to launch the external procedure agent, thus allowing you to discover the corresponding process ID (PID). In a fresh SQL*Plus session, you can run it as follows:

```
SQL> EXEC DEBUG_EXTPROC.startup_extproc_agent
```

This causes an extproc process to launch; you can find its PID using ps -ef or pgrep extproc.

Why do I say the DEBUG_EXTPROC package is optional? Because you can also launch the agent by running any old external procedure, or, if you happen to be using multithreaded agents, the process will be prespawned, and you can probably figure out the PID without breaking a sweat.

At any rate, armed with the PID, you can start the debugger and attach to the running process:

```
OS> gdb $ORACLE_HOME/bin/extproc pid
```

When I first tried this, I got a "permission denied" error, which I cured by logging in as the oracle account.

I then set a breakpoint on the pextproc symbol, per the instructions in the file *dbgextp.sql*. Next, in my SQL*Plus session, I invoked my external procedure using:

```
SQL> CALL shell(NULL);
```

I issued a GDB continue, and extproc promptly hit the pextproc breakpoint. Next, I executed a GDB share command so the debugger would read the symbols in my just-loaded external shared library; finally, I was able to set a breakpoint on the extprocsh external procedure, issue a continue, and boom—I'm in my code! It worked pretty well after that, allowing me to step through each line of my code, examine variables, and so on.

I found that debugging external procedures with Cygwin's GDB on Microsoft platforms required the following adjustments:

- I had to modify the listener service in the Windows Control Panel to execute under the authority of my own user account rather than under that of "Local System."
- Instead of ps -ef, I used Microsoft's *tasklist.exe* program (or the Windows Task Manager) to obtain the extproc PID.
- To view the external procedure's source code during the debugging session, I found that I needed to launch GDB from the directory containing its source file (there is probably another way to do this).

On Solaris, I performed my tests using a 64-bit build of GDB 6.3 on Solaris 2.8. On my Windows XP machine, I used Cygwin's GDB 6.3 binary with no problems, but was not able to get the MinGW GDB 5.2.1 binary to work.

Maintaining External Procedures

Here are some assorted bits of information that will assist you in creating, debugging, and managing external procedures.

Dropping Libraries

The syntax for dropping a library is simply:

```
DROP LIBRARY library_name;
```

The Oracle user who executes this command must have the DROP LIBRARY or DROP ANY LIBRARY privilege.

Oracle does not check dependency information before dropping the library. This fact is useful if you need to change the name or location of the shared object file to which the library points. You can just drop it and rebuild it, and any dependent routines will continue to function. (More useful, perhaps, would be a requirement that you use a DROP LIBRARY FORCE command, but such an option does not exist.)

Before you drop the library permanently, you may wish to look in the DBA_DEPENDENCIES view to see if any PL/SQL module relies on the library.

Data Dictionary

There are a few entries in the data dictionary that help manage external procedures. Table 28-3 shows the USER_ versions of the dictionary tables, but note that there are corresponding entries for DBA_ and ALL_.

Table 28-3. Data dictionary views for external procedures

To answer the question...	Use this view	Example

What libraries have I created?	USER_LIBRARIES	`SELECT *`
		`FROM user_libraries;`

What stored PL/SQL programs use the xyz library in a call spec?	USER_DEPENDENCIES	`SELECT *`
		`  FROM user_dependencies`
		` WHERE referenced_name = 'XYZ';`

What external procedure agents (both dedicated and multithreaded) are currently running?	V$HS_AGENT	`SELECT * FROM V$HS_AGENT`
		` WHERE UPPER(program) LIKE 'EXTPROCS%'`

What Oracle sessions are using which agents?	V$HS_SESSION	`SELECT s.username, h.agent_id`
		`  FROM V$SESSION s, V$HS_SESSION h`
		` WHERE s.sid = h.sid;`

Rules and Warnings

As with almost all things PL/SQL, external procedures come with an obligatory list of cautions:

- While the mode of each formal parameter (IN, IN OUT, OUT) may have certain restrictions in PL/SQL, C does not honor these modes. Differences between the PL/SQL parameter mode and the usage in the C module cannot be detected at compile time, and could also go undetected at runtime. The rules are what you would expect: don't assign values to IN parameters, don't read OUT parameters, always assign values to IN OUT and OUT parameters, and always return a value of the appropriate datatype.
- Modifiable INDICATORs and LENGTHs are always passed by reference for IN OUT, OUT, and RETURN. Unmodifiable INDICATORs and LENGTHs are always passed by value unless you specify BY REFERENCE. However, even if you pass INDICATORs or LENGTHs for PL/SQL variables by reference, they are still read-only parameters.
- Although you can pass up to 128 parameters between PL/SQL and C, if any of them are floats or doubles, your actual maximum will be lower. How much lower depends on the operating system.

- If you use the multithreaded agent feature introduced in Oracle Database 10*g*, there are special additional restrictions on your programs. All the calls you invoke from the C program must be thread-safe. In addition, you'll want to avoid using global C variables. Even in the nonthreaded version, globals may not behave as expected due to "DLL caching" by the operating system.
- Your external procedure may not perform DDL commands, begin or end a session, or control a transaction using COMMIT or ROLLBACK. (See Oracle's *PL/SQL User's Guide and Reference* for a list of unsupported OCI routines.)