

This chapter is provided on an "as is" basis as part of the Apress Beta Book Program. Please note that content is liable to change before publication of the final book, and that neither the author(s) nor Apress will accept liability for any loss or damage caused by information contained.

Copyright (c) 2005. For further information email [support@apress.com](mailto:support@apress.com)

All rights reserved. No part of this work may be reproduced in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

## CHAPTER 5

# Oracle Processes

We have reached the last piece of the puzzle. We've investigated the database and the set of physical files that constitute a database. In covering the memory used by Oracle, we have looked at one half of an instance. The last remaining architectural issue is the set of processes that constitute the other half of the instance. Some of these processes, such as the database block writer (**DBWn**), and the log writer (**LGWR**) have cropped up already. Here, we will take a closer look at the function of each, what they do and why they do it. When we talk of 'process' here, it will be synonymous with 'thread' on operating systems where Oracle is implemented with threads such as Windows. So, for example, when we talk about the **DBWn** process, the equivalent on Windows is the **DBWn** thread. In the context of this chapter, we will use the term process to cover both process and threads. If you are using an implementation of Oracle that is multi-process, such as you see on Unix – the term is totally appropriate. If you are using a single process implementation of Oracle such as you see on Windows, the term process will actually mean "thread within the Oracle process".

There are three classes of processes in an Oracle instance:

- \* *Server Processes* – These perform work based on a client's request. We have already looked at dedicated, and shared servers to some degree. These are the server processes.
- \* *Background Processes* – These are the processes that start up with the database, and perform various maintenance tasks, such as writing blocks to disk, maintaining the online redo log, cleaning up aborted processes, and so on.
- \* *Slave Processes* – These are similar to background processes but they are processes that perform extra work on behalf of either a background or a server process.

We will take a look at the processes in each of these three classes to see how they fit into the picture.

## Server Processes

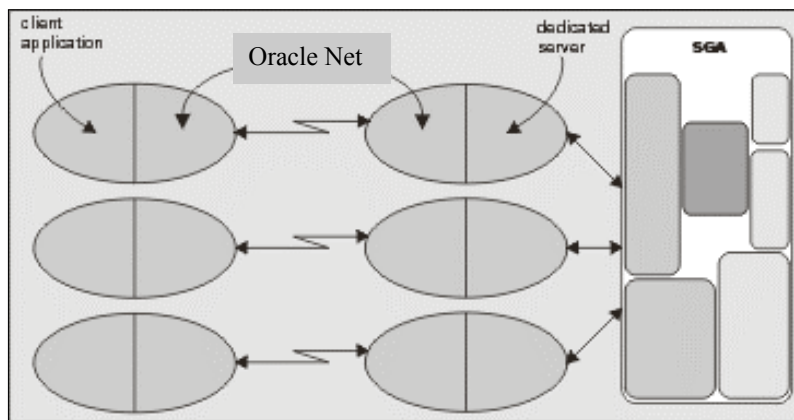
We briefly touched on two connection types to Oracle, specifically:

- \* **Dedicated Server:** whereby you get a dedicated process on the server for your connection. There is a one to one mapping between a connection to the database and a process or thread. Note that a connection is not a session! A connection is simply a physical path from your client to the server. We'll cover that in more detail below.
- \* **Shared Server:** whereby you share a pool of server processes spawned and managed by the Oracle instance. Your connection is to a database dispatcher, not a dedicated server process created just for your connection.

Both dedicated and shared servers have the same jobs – they process all of the SQL you give to them. When you submit a `SELECT * FROM EMP` query to the database, it is an Oracle dedicated/shared server that will parse the query, and place it into the shared pool (or find it in the shared pool already, hopefully). It is this process that will come up with the query plan if necessary. It is this process that will execute the query plan, perhaps finding the necessary data in the buffer cache, or reading the data from disk into the buffer cache. These server processes are the ‘work horse’ processes. Many times, you will find these processes to be the highest consumers of CPU time on your system, as they are the ones that do your sorting, your summing, your joining – pretty much everything.

### Dedicated Server Connections

In dedicated server mode, there will be a one-to-one mapping between a client connection and a server process (or thread, as the case may be). If you have 100 dedicated server connections on a UNIX machine, there will be 100 processes executing on their behalf. Graphically it would look like this:



[Insert 2433f0213scrap.gif](#)

Figure 2-13. Typical Dedicated Server Connection.

Your client application will have Oracle libraries linked into it. These libraries provide the **Application Program Interface (API)** that you need in order to talk to the database. These APIs know how to submit a query to the database, and process the cursor that is returned. They know how to bundle your requests into network calls that the dedicated server will know how to un-bundle. This piece of the picture is called *Oracle Net*, in prior releases you

might know this piece of software as *SQL\*Net* or *Net8*. This is the networking software/protocol that Oracle employs to allow for client server processing (even in a n-tier architecture, there is a client server program lurking). Oracle employs this same architecture even if Oracle Net is not technically involved in the picture. That is, when the client and server are on the same machine this two-process (also known as two-task) architecture is employed. This architecture provides two benefits:

- \* *Remote Execution* – It is very natural for the client application to be executing on a machine other than the database itself.
- \* *Address Space Isolation* – The server process has read-write access to the SGA. An errant pointer in a client process could easily corrupt data structures in the SGA, if the client process and server process were physically linked together.

Earlier in Chapter 2 we saw how these dedicated servers are ‘spawned’ or created by the Oracle Listener process. We won’t cover that process again, but rather we’ll quickly look at what happens when the listener is not involved. The mechanism is much the same as it was with the listener, but instead of the listener creating the dedicated server via a `fork()/exec()` in UNIX and an *IPC* (Inter Process Communication) call in Windows, it is the client process itself that creates it.

---

NOTE: There are many variants of the `fork()` and `exec()` calls such as `vfork()`, `execve()` and so on. The call used by Oracle may vary by OS and implementation but the net effect is the same. `Fork()` creates a new process that is a clone of the parent process and on Unix is the only way to create new process. `Exec()` loads a new program image over the existing program image in memory, thus starting a new program. So, SQL\*Plus can “fork” (copy itself) and then “exec” the Oracle binary, overlaying the copy of itself with this new program.

---

We can see this clearly on UNIX when we run the client and server on the same machine:

```
ops$tkyte@ORA10G> select a.spid dedicated_server,
2      b.process clientpid
3  from v$process a, v$session b
4  where a.addr = b.paddr
5     and b.sid = (select sid from v$mystat where rownum=1)
6  /
```

DEDICATED\_SE CLIENTPID

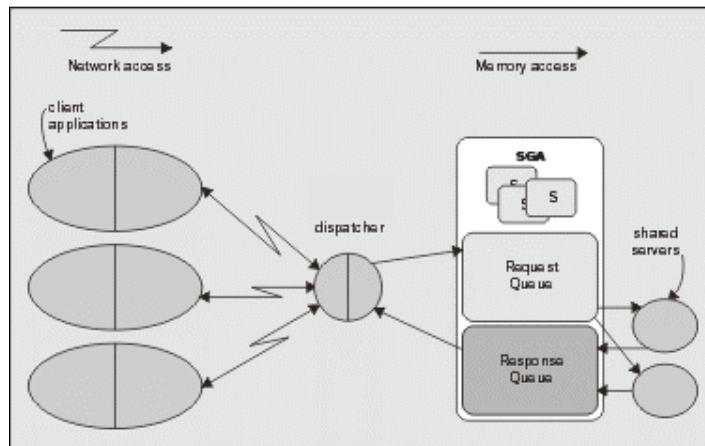
```
-----
5114      5112
```

```
ops$tkyte@ORA10G> !/bin/ps -p 5114 5112
PID TTY    STAT  TIME COMMAND
5112 pts/1   R     0:00 sqlplus
5114 ?      S     0:00 oracleora10g (DESCRIPTION=(LOCAL=YES)..))
```

Here, I used a query to discover the *Process ID* (PID) associated with my dedicated server (the SPID from `V$PROCESS` is the operating system process ID of the process that was being used during the execution of that query).

## Shared Server Connections

Let's now take a look at the other type of Server process, the shared server process, in more detail. These types of connections mandate the use of Oracle Net even if the client and server are on the same machine – you cannot use shared server without using the Oracle TNS listener. As we described earlier, the client application will connect to the Oracle TNS listener and will be redirected or handed off to a dispatcher. The dispatcher will act as the conduit between the client application and the shared server process. Following is a diagram of the architecture of a shared server connection to the database:



*[Insert 2433f0214scrap.gif](#)*

Figure 2-14. Typical Shared Server Connection

Here, we can see that the client applications, with the Oracle libraries linked in, will be physically connected to a dispatcher process. We may have many dispatchers configured for any given instance but it is not uncommon to have just one dispatcher for many hundreds, to thousands, of users. The dispatcher is simply responsible for receiving inbound requests from the client applications, and putting them into the SGA in a request queue. The first available shared server process, which is basically the same as a dedicated server process, will pick up the request from the queue, and attach the UGA of the associated session (the 'S' boxes depicted in the above diagram). The shared server will process that request and place any output from it into the response queue. The dispatcher is constantly monitoring the response queue for results, and transmits them back to the client application. As far as the client is concerned, it cannot really tell if it is connected via a dedicated server, or a shared connection – they appear to be the same. Only at the database level is the difference apparent.

## Connections versus Sessions

It surprises many people to discover that a connection is not synonymous with a session. In most people's eyes they are, but the reality is they do not have to be. A connection may have zero one or more sessions established on it. Each session is separate and independent, even though they all share the same physical connection to the database. A commit in one session

does not affect any other session on that connection. In fact, each session using that connection could use different identities!

In Oracle, a connection is simply a physical circuit between your client process and the database instance, a network connection most commonly. The connection may be to a dedicated server process or to a dispatcher. As stated, a connect may have zero or more sessions – meaning a connection may exist with no corresponding sessions. Additionally, a session *may or may not* have a connection. Utilizing advanced Oracle Net features such as connection pool, a physical connection may be dropped by a client, leaving the session intact (but idle). When the client wants to perform some operation in that session, the client would re-establish the physical connection. So, let's define these terms:

- \* *Connection*: A connection is a physical path from a client to an Oracle instance. A connection is either established over a network or might be established over an IPC mechanism. A connection is typically between a client process and either a dedicated server or a dispatcher, however using Oracle's Connection Manager (CMAN), a connection may be between a client and CMAN and CMAN and the database as well. CMAN is out of scope for the purposes of this book, however the Oracle Net Services Administrators Guide freely available on [otn.oracle.com](http://otn.oracle.com) covers it in some detail.
- \* *Session*: A session is a logical entity that exists in the database. It is what most people would commonly think of when they think of a "database connection". It is your session in the server, where you would execute your SQL, commit transactions, run stored procedures.

So, how can we see some of this, and how common is it for a connection to have more than one session? Well, we can once again use SQL\*Plus to not only see this in action, but also to recognize that this could be a very common thing indeed. We'll simply use the AUTOTRACE command and discover that we have two sessions! Over a single connection, using a single process – we'll establish two sessions:

```
ops$tkyte@ORA10G> select username, sid, serial#, server, paddr, status
2   from v$session
3  where username = USER
4  /
```

USERNAME	SID	SERIAL#	SERVER	PADDR	STATUS
OPS\$TKYTE	153	3196	DEDICATED	AE4CF614	ACTIVE

Now, that shows right now that I had one session, if you are running this test yourself, just note the sessions you have if more than one. I currently have one dedicated server connected session going. The PADDR column is the process address of my sole dedicated server. By simply turning on AUTOTRACE to see the statistics of statements I execute in SQL\*Plus:

```
ops$tkyte@ORA10G> set autotrace on statistics
ops$tkyte@ORA10G> select username, sid, serial#, server, paddr, status
2   from v$session
3  where username = USER
4  /
```

USERNAME	SID	SERIAL#	SERVER	PADDR	STATUS
----------	-----	---------	--------	-------	--------

```

-----
OPPS$TKYTE 151    1511 DEDICATED AE4CF614 INACTIVE
OPPS$TKYTE 153    3196 DEDICATED AE4CF614 ACTIVE

```

#### Statistics

```

-----
0 recursive calls
0 db block gets
0 consistent gets
0 physical reads
0 redo size
756 bytes sent via SQL*Net to client
508 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
2 rows processed
ops$tkyte@ORA10G> set autotrace off

```

We can see that I now have two sessions – but both are using the same single dedicated server process as evidenced by them both having the same PADDR value. You can confirm in the operating system that no new processes were created, we are using a single process, a single connection for both sessions. Note that one of the sessions is ACTIVE, the original session. That makes sense – it is running the query to show this information so of course it is active, but that INACTIVE session, what is that one for? That is the AUTOTRACE session, it's job is to “watch” our real session and report on what it does. When you enable AUTOTRACE in SQL\*Plus, SQL\*Plus will do the following when you execute DML operations (INSERT, UPDATE, DELETE, SELECT, MERGE):

- \* It will create a new session using the current connection if the secondary session does not already exist.
- \* It will ask this new session to query the V\$SESSTAT view to remember the beginning statistic values for the session we will run the DML in. Very much like the function the watch\_stat.sql script performed for us in Chapter 4.
- \* It will run the DML operation in the original session
- \* Upon completion of that DML statement, SQL\*Plus will request the other session to query V\$SESSTAT again and produce the report you see above – showing the difference in the statistics for the session that executed the DML.

If you set AUTOTRACE off, SQL\*Plus will terminate this additional session and you will no longer see it in V\$SESSION. A question you might ask is why does SQL\*Plus do this trick? The answer is fairly straightforward, for the same reason we used another SQL\*Plus session in Chapter 4 to monitor memory and temporary space usage – if we used a single session to monitor memory usage, we would be using memory to do the monitoring. By observing the statistics in a single session you necessarily would change those statistics. If SQL\*Plus used a single session to report on the number of IO's performed, how many bytes were transferred over the network, how many sorts happened – the queries used to find these details would be adding to the mix themselves. They could be sorting, performing IO,

transferring data over the network (one would assume they would!). Hence, we need to use another session in order to measure correctly.

So far, we've seen a connection with 1 or two sessions, now we'd like to use SQL\*Plus to see a connection with no session. That one is pretty easy, in this same session from above, simply type the "misleading" command DISCONNECT:

```
ops$tkyte@ORA10G> disconnect
Disconnected from Oracle Database 10g Enterprise Edition Release 10.1.0.3.0 - Production
With the Partitioning, OLAP and Data Mining options
ops$tkyte@ORA10G>
```

That command should be called "destroy\_all\_sessions" instead of disconnect technically, since we haven't really disconnected physically – we have however closed all of our sessions. If you open another session using some other user account and query (replacing OPS\$TKYTE with your account name of course):

```
sys@ORA10G> select * from v$session where username = 'OPS$TKYTE';
no rows selected
```

You can see that we have no sessions – but we still have a process, a physical connection:

```
sys@ORA10G> select username, program
2   from v$process
3  where addr = hextoraw('AE4CF614');
```

USERNAME	PROGRAM
tkyte	oracle@localhost.localdomain (TNS V1-V3)

So, here we have a "connection" with no sessions associated with it. We can use the also misnamed SQL\*Plus command "connect" to create a new session in this existing process (the command "connect" might be better named as "create\_session"):

```
ops$tkyte@ORA10G> connect /
Connected.
```

```
ops$tkyte@ORA10G> select username, sid, serial#, server, paddr, status
2   from v$session
3  where username = USER
4  /
```

USERNAME	SID	SERIAL#	SERVER	PADDR	STATUS
OPS\$TKYTE	150	233	DEDICATED	AE4CF614	ACTIVE

So, notice that we have the same PADDR, we are using the same physical connection, but that we have (potentially) a different SID or session id. I say potentially because we could get assigned the same SID, it just depends on whether other people logged in while we were logged out and if the original SID we had was available.

So far, these tests were performed using a dedicated server connection, the PADDR was the process address of our dedicated server process. What happens if we use shared server however. Well, if I log in using a shared server and in that session query:

```
ops$tkyte@ORA10G> select a.username, a.sid, a.serial#, a.server,
2   a.paddr, a.status, b.program
```

```

3  from v$session a left join v$process b
4    on (a.paddr = b.addr)
5  where a.username = 'OPS$TKYTE'
6  /

```

```

USERNAME  SID SERIAL# SERVER  PADDR  STATUS
-----
PROGRAM
-----

```

```

OPS$TKYTE 150   261 SHARED  AE4CF118 ACTIVE
oracle@localhost.localdomain (S000)

```

Our shared server connection is associated with a process – the PADDR is there and we can join to V\$PROCESS to pick up the name of this process, in this case – we see it is a shared server, the text (S000) tells us that (below we'll see the names of all of the Oracle processes). But, if we use another session to query this same bit of information, while leaving our shared server session idle, we'd see something like this:

```

sys@ORA10G> select a.username, a.sid, a.serial#, a.server,
2             a.paddr, a.status, b.program
3  from v$session a left join v$process b
4    on (a.paddr = b.addr)
5  where a.username = 'OPS$TKYTE'
6  /

```

```

USERNAME  SID SERIAL# SERVER  PADDR  STATUS
-----
PROGRAM
-----

```

```

OPS$TKYTE 150   261 NONE    AE4CEC1C INACTIVE
oracle@localhost.localdomain (D000)

```

Notice that our PADDR is different and the name of the process we are associated with has changed as well. Our idle shared server connection is now associated with a dispatcher – D000. Hence we have yet another method for observing multiple sessions pointing to a single process – a dispatcher could have hundreds or even thousands of sessions pointing to it. Now an interesting attribute of shared server connections is that the shared server process we use from call to call can change. If I were the only one using this system (as I am for these tests), running that query over and over as OPS\$TKYTE would tend to produce the same PADDR of AE4CF118 over and over. However, if I were to open up more shared server connections and start to use that shared server in other sessions – I might notice that the shared server I use varies. Consider this example. I'll query up my current session information, showing the shared server I am using and then in another shared server session I'll perform a long running operation (I'll monopolize that shared server). When I ask the database what shared server I'm using again, I will most likely see a different one (if the original one is off servicing the other session). In the following example, the code in bold represents a second SQL\*Plus session that was connected via shared server:

```

ops$tkyte@ORA10G> select a.username, a.sid, a.serial#, a.server,
2             a.paddr, a.status, b.program
3  from v$session a left join v$process b
4    on (a.paddr = b.addr)
5  where a.username = 'OPS$TKYTE'

```



```

USERNAME  SID SERIAL# SERVER  PADDR  STATUS
-----
PROGRAM
-----

```

```

OPS$TKYTE 150 261 SHARED AE4CF118 ACTIVE
oracle@localhost.localdomain (S000)

```

```

sys@ORA10G> connect system/manager@shared_server.us.oracle.com
Connected.
system@ORA10G> exec dbms_lock.sleep(20)

```

```

ops$tkyte@ORA10G> select a.username, a.sid, a.serial#, a.server,
2      a.paddr, a.status, b.program
3      from v$session a left join v$process b
4      on (a.paddr = b.addr)
5      where a.username = 'OPS$TKYTE'
6 /

```

```

USERNAME  SID SERIAL# SERVER  PADDR  STATUS
-----
PROGRAM
-----

```

```

OPS$TKYTE 150 261 SHARED AE4CF614 ACTIVE
oracle@localhost.localdomain (S001)

```

Notice how the first time we queried, we were using S000 as the shared server. Then in another session we executed a long running statement that monopolized the shared server which just happened to be S000 this time. The first non-busy shared server is the one that gets assigned the work to do and in this case no one else was asking to use the S000 shared server so the DBMS\_LOCK command took it. Now, when we queried again in our session, since the S000 shared server was busy, we got assigned to another shared server process. It is interesting to note that the parse of a query (returns no rows yet) could be processed by shared server S000, the first of the first row by S001, the fetch of the second row by S002 and the closing of the cursor by S003. That is, an individual statement might be processed bit by bit by many shared servers.

So, what we have seen in this section is that a connection, a physical pathway from a client to a database instance may have zero, one or more sessions established on it. We have seen one use case of that – SQL\*Plus's AUTOTRACE facility. Many other tools employ this ability as well – for example Oracle Forms uses multiple sessions on a single connection to implement their debugging facilities. The n-tier proxy authentication feature of Oracle, used to provide end to end identification of users from the browser to the database makes use heavily of the concept of a single connection with multiple sessions – but each session there would be using a potentially different user account. We have seen that sessions can use many processes over time, especially in a shared server environment. Also, if you are using connection pooling with Oracle Net – your session might not be associated with any process at all, the client would drop the connection after an idle time and re-establish it transparent upon activity. In short, there is a many to many relationship between connections and

sessions. The most common case, the one most of us see day to day however is a one to one relationship between a dedicated server and a single session.

## Dedicated Server versus Shared Server

Before we continue onto the rest of the processes, we'll discuss why there are two modes of connections, and when one might be more appropriate over the other. Dedicated server mode is by far the most common method of connection to the Oracle database for all SQL-based applications. It is the easiest to set up and provides the easiest way to establish connections. It requires little to no configuration. Shared server setup and configuration, while not difficult, is an extra step. The main difference between the two is not, however, in their set up. It is in their mode of operation. With dedicated server, there is a one-to-one mapping between client connections, and server process. With shared server there is a many-to-one relationship – many clients to a shared server. As the name implies, shared server is a shared resource, whereas the dedicated server is not. When using a shared resource, you must be careful not to monopolize it for long periods of time. As we saw above using a simple “DBMS\_LOCK.SLEEP(20)” in one session would monopolize a shared server process for 20 seconds, monopolizing these shared server resources can lead to a system that appears to be hung. In Figure 2-14. Typical Shared Server Connection a couple of pages back, I have two shared servers depicted. If I have three clients, and all of them attempt to run a 45-second process more or less at the same time, two of them will get their response in 45 seconds; the third will get its response in 90 seconds. This is rule number one for shared server – make sure your transactions are short in duration. They can be frequent, but they should be short (as characterized by OLTP systems). If they are not, you will get what appears to be a total system slowdown due to shared resources being monopolized by a few processes. In extreme cases, if all of the shared servers are busy, the system will appear to be hung.

So, shared server is highly appropriate for an OLTP system characterized by short, frequent transactions. In an OLTP system, transactions are executed in milliseconds – nothing ever takes more than a fraction of a second. Shared server on the other hand is highly inappropriate for a data warehouse. Here, you might execute a query that takes one, two, five, or more minutes. Under shared server, this would be deadly. If you have a system that is 90 percent OLTP and 10 percent ‘not quite OLTP’, then you can mix and match dedicated servers and shared server on the same instance. In this fashion, you can reduce the number of processes on the machine dramatically for the OLTP users, and make it so that the ‘not quite OLTP’ users do not monopolize their shared servers.

So, what are the benefits of shared server, bearing in mind that you have to be somewhat careful about the transaction types you let use it? Shared server does three things for us, mainly:

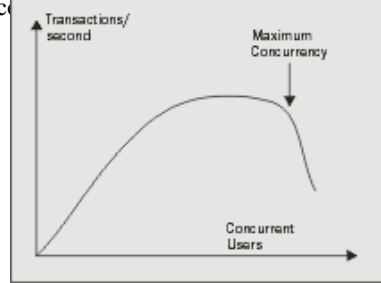
*Reduces the number of OS processes/threads:*

On a system with thousands of users, the OS may quickly become overwhelmed in trying to manage thousands of processes. In a typical system, only a fraction of the thousands of users are concurrently active at any point in time. For example, I've worked on systems recently with 5000 concurrent users. At any one point in time, at most 50 were active. This system would work effectively with 50 shared server processes, reducing the number of processes the operating system has to manage by two orders of magnitude (100 times). The operating system can now, to a large degree, avoid context switching.

*Allows you to artificially limit the degree of concurrency:*

As a person who has been involved in lots of benchmarks, the benefits of this are obvious to me. When running benchmarks, people frequently ask to run as many users as possible

until the system breaks. One of the outputs of these benchmarks is always a chart that shows the number of concurrent users vs. transactions per second:



*Insert 2433f0215scrap.gif*

*Figure 2-15. Concurrent Users vs Transactions Per Second.*

Initially, as you add concurrent users, the number of transactions goes up. At some point however, adding additional users does not increase the number of transactions you can perform per second – it tends to go flat. The throughput has peaked and now response time starts to go up (we are doing the same number of TPS, but the end users are observing slower response times). As you continue adding users, you will find that the throughput will actually start to decline. The concurrent user count before this drop off is the maximum degree of concurrency you want to allow on the system. Beyond this point, the system is becoming flooded and the queues are forming to perform work. Much like a backup at a tollbooth, the system can no longer keep up. Not only does response time rise dramatically at this point, but throughput from the system falls as well. If we limit the maximum concurrency to the point right before this drop, we can sustain maximum throughput, and minimize the increase in the response time for most users. Shared server allows us to limit the maximum degree of concurrency on our system to this number.

*Reduces the memory needed on the system:*

This is one of the most highly touted reasons for using shared server – it reduces the amount of required memory. It does, but not as significantly as you might think, especially given the new automatic PGA memory management discussed in Chapter 4, where workareas are allocated to a process, used and releases – and their size varies based on the concurrent workload. So, this was a fact that was *more true* in older releases of Oracle but is not as meaningful today. Also, remember that when we use shared server, the UGA is located in the SGA. This means that when switching over to shared server, you must be able to accurately determine your expected UGA memory needs, and allocate appropriately in the SGA, via the `LARGE_POOL`. So, the SGA requirements for the shared server configuration are typically very large. This memory must be pre-allocated and thus, can only be used by the database. Contrast this with dedicated server, where anyone can use any memory not allocated to the SGA. So, if the SGA is much larger due to the UGA being located in it, where does the memory savings come from? It comes from having that many less PGAs allocated. Each dedicated/shared server has a PGA. This is process information. It is sort areas, hash areas, and other process related structures. It is this memory need that you are

removing from the system by using shared server. If you go from using 5000 dedicated servers to 100 shared servers, it is the cumulative sizes of the 4900 PGAs you no longer need, that you are saving with shared server.

Of course, the final reason to use shared server is when you have no choice. If you want to use Oracle Net Connection pool for example, you must use shared server. There are many other advanced connection features that require the use of shared server. If you want to use database link concentration between databases, for example, then you must be using shared server for those connections.

## A Recommendation

Unless your system is overloaded, or you need to use shared server for a specific feature, a dedicated server will probably serve you best. A dedicated server is simple to set up, and makes tuning easier. There are certain operations that must be done in a dedicated server mode so every database will have either both, or just a dedicated server set up.

On the other hand, if you have a very large user community and *know* that you will be deploying with shared server, I would urge you to *develop and test* with shared server. It will increase your likelihood of failure if you develop under just a dedicated server and never test on shared server. Stress the system, benchmark it, make sure that your application is well behaved under shared server. That is, make sure it does not monopolize shared servers for too long. If you find that it does so during development, it is much easier to fix than during deployment. You can utilize features such as the Advanced Queues (AQ) to turn a long running process into an apparently short one, but you have to *design* that into your application. These sorts of things are best done when you are developing. Also, there have historically been differences between the feature set available to shared server connections versus dedicated server connections. We already discussed the lack of automatic PGA memory management in Oracle 9i for example – but also in the past things as basic as a hash join between two tables were not available in shared server connections.

---

NOTE: If you are already using a connection-pooling feature in your application (for example, you are using the J2EE connection pool), and you have sized your connection pool appropriately, using shared server will only be a performance inhibitor. You already sized your connection pool to cater for the number of concurrent connections that you will get at any point in time – you want each of those connections to be a direct dedicated server connection. Otherwise, you just have a connection pooling feature connecting to yet another connection pooling feature.

---

## Background Processes

The Oracle instance is made up of two things: the SGA and a set of background processes. The background processes perform the mundane maintenance tasks needed to keep the database running. For example, there is a process that maintains the block buffer cache for us, writing blocks out to the data files as needed. There is another process that is responsible for copying an online redo log file to an archive destination as it fills up. There is another process responsible for cleaning up after aborted processes, and so on. Each of these processes is pretty focused on its job, but works in concert with all of the others. For example, when the process responsible for writing to the log files fills one log and goes to the next, it will notify the process responsible for archiving that full log file that there is work to be done.

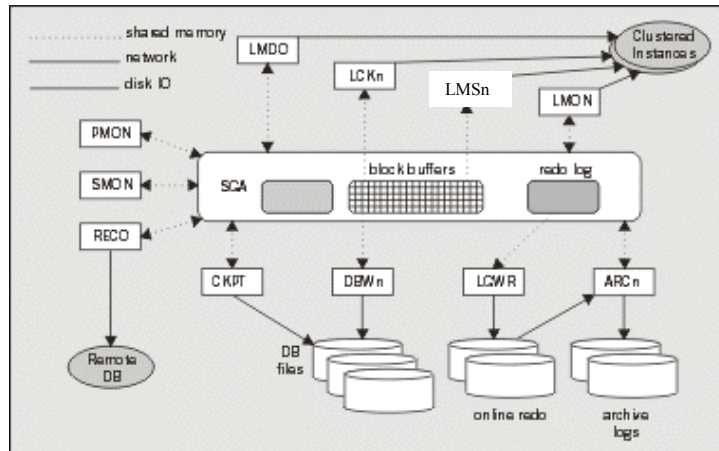
There are two classes of background processes: those that have a focused job to do (as we have just described), and those that do a variety of other jobs. For example, there is a background process for the internal job queues in Oracle. This process monitors the job queues and runs whatever is inside of them. In many respects, it resembles a dedicated server process, but without a client connection. We will look at each of these background processes now, starting with the ones that have a focused job, and then going into the ‘all-purpose’ processes.

## Focused Background Processes

Thomas Kyte

**Comment:** Need fix for LMSn in final graphic.

The following diagram depicts the Oracle background processes that have a focused purpose:



*Insert 2433f0216scrap.gif*

Figure 2-16. Insert figure caption here.

You may not see all of these processes when you start your instance, but the majority of them will be present. You will only see **ARCn** (the archiver) if you are in Archive Log Mode and have enabled automatic archiving. You will only see the **LMD0**, **LCKn**, **LMON**, and **LMSn** (more details on those processes below) processes if you are running Oracle Real Application Clusters, a configuration of Oracle that allows many instances on different machines in a cluster to mount and open the same physical database. For the sake of clarity, missing from the above picture are the shared server dispatcher (**Dnnn**) and shared server (**Snnn**) processes. As we just covered them in some detail, I left them out in order to make the diagram a little more readable. The previous figure depicts what you might ‘see’ if you started an Oracle instance, and mounted and opened a database. For example, on my Linux system, after starting the instance up, I have the following processes:

```
$ ps -aef | grep 'ora_*_ora10g$'
ora10g 5892 1 0 16:17 ? 00:00:00 ora_pmon_ora10g
ora10g 5894 1 0 16:17 ? 00:00:00 ora_mman_ora10g
ora10g 5896 1 0 16:17 ? 00:00:00 ora_dbw0_ora10g
ora10g 5898 1 0 16:17 ? 00:00:00 ora_lgwr_ora10g
ora10g 5900 1 0 16:17 ? 00:00:00 ora_ckpt_ora10g
```

ora10g	5902	1	0	16:17	?	00:00:00	ora_smon_ora10g
ora10g	5904	1	0	16:17	?	00:00:00	ora_reco_ora10g
ora10g	5906	1	0	16:17	?	00:00:00	ora_cjq0_ora10g
ora10g	5908	1	0	16:17	?	00:00:00	ora_d000_ora10g
ora10g	5910	1	0	16:17	?	00:00:00	ora_s000_ora10g
ora10g	5916	1	0	16:17	?	00:00:00	ora_arc0_ora10g
ora10g	5918	1	0	16:17	?	00:00:00	ora_arc1_ora10g
ora10g	5920	1	0	16:17	?	00:00:00	ora_qmnc_ora10g
ora10g	5922	1	0	16:17	?	00:00:00	ora_mmon_ora10g
ora10g	5924	1	0	16:17	?	00:00:00	ora_mmm1_ora10g
ora10g	5939	1	0	16:28	?	00:00:00	ora_q000_ora10g

They mostly correspond to the processes depicted above, there are more listed here than in the picture but that is more for clarity in the picture than anything else. It is interesting to note the naming convention used by these processes. The process name starts with **ora\_**. It is followed by four characters representing the actual name of the process, and then by **\_ora10g**. As it happens, my **ORACLE\_SID** (site identifier) is **ora10g**. On UNIX, this makes it very easy to identify the Oracle background processes and associate them with a particular instance (on Windows, there is no easy way to do this, as the backgrounds are threads in a larger, single process). What is perhaps most interesting, but not readily apparent from the above, is that *they are all really the same exact binary*. Search as hard as you like but you will not find the **arc0** binary executable on disk anywhere. You will not find **LGWR** or **DBW0**. These processes are all really **oracle** (that's the name of the binary executable that is run). They just alias themselves upon start-up in order to make it easier to identify which process is which. This enables a great deal of object code to be efficiently shared on the UNIX platform. On Windows, this is not nearly as interesting, as they are just threads within the process – so of course they are one big binary.

Let's now take a look at the function performed by each process, starting with the primary Oracle background processes.

### PMON – The Process Monitor

This process is responsible for cleaning up after abnormally terminated connections. For example, if your dedicated server 'fails' or is killed for some reason, **PMON** is the process responsible for releasing your resources. **PMON** will rollback uncommitted work, release locks, and free SGA resources allocated to the failed process.

In addition to cleaning up after aborted connections, **PMON** is responsible for monitoring the other Oracle background processes and restarting them if necessary (and if possible). If a shared server or a dispatcher fails (crashes) **PMON** will step in, and restart another one (after cleaning up for the failed process). **PMON** will watch all of the Oracle processes, and either restart them or terminate the instance as appropriate. For example, it is appropriate to restart the instance in the event the database log writer process, **LGWR**, fails. This is a serious error and the safest path of action is to terminate the instance immediately and let normal recovery fix up the data. This is a very rare occurrence and should be reported to Oracle support immediately.

The other thing **PMON** does for the instance is to register it with the Oracle TNS listener. When an instance starts up, the **PMON** process polls the well-known port address (unless directed otherwise) to see whether or not a listener is up and running. The well-known/default port used by Oracle is 1521. Now, what happens if the listener is started on some different port? In this case the mechanism is the same, except that the listener address needs to be explicitly specified by the **LOCAL\_LISTENER** parameter setting. If the listener is

started, **PMON** communicates with the listener and passes to it relevant parameters, such as the service name and load metrics of the instance.

### **SMON – The System Monitor**

**SMON** is the process that gets to do all of the jobs no one else wants to do. It is a sort of ‘garbage collector’ for the database. Some of the jobs it is responsible for include:

- \* *Temporary space clean up* – With the advent of ‘true’ temporary tablespaces, this job has lessened, but has not gone away. For example, when building an index, the extents allocated for the index during the creation are marked as **TEMPORARY**. If the **CREATE INDEX** session is aborted for some reason, **SMON** is responsible for cleaning them up. There are other operations that create temporary extents that **SMON** would be responsible for as well.
- \* *Crash recovery* – **SMON** is responsible for performing crash recovery of a failed instance, upon restart.
- \* *Coalescing free space* – If you are using dictionary-managed tablespaces, **SMON** is responsible for taking extents that are free in a tablespace and contiguous with respect to each other, and coalescing them into one ‘bigger’ free extent. This occurs only on dictionary managed tablespace with a default storage clause that has **pctincrease** set to a non-zero value.
- \* *Recovering transactions active against unavailable files* – This is similar to its role during database startup. Here **SMON** recovers failed transactions that were skipped during instance/crash recovery due to a file(s) not being available to recover. For example, the file may have been on a disk that was unavailable or not mounted. When the file does become available, **SMON** will recover it.
- \* *Instance recovery of failed node in RAC* – In an Oracle Real Application Clusters configuration, when a node in the cluster goes down (the machine fails), some other node in the instance will open that failed node’s redo log files, and perform a recovery of all data for that failed node.
- \* *Cleans up OBJ\$* – **OBJ\$** is a low-level data dictionary table that contains an entry for almost every object (table, index, trigger, view, and so on) in the database. There are many times entries in here that represent deleted objects, or objects that represent ‘not there’ objects, used in Oracle’s dependency mechanism. **SMON** is the process that removes these no longer needed rows.
- \* *Shrinks rollback segments* – **SMON** is the process that will perform the automatic shrinking of a rollback segment to its optimal size, if it is set.
- \* *‘Offlines’ rollback segments* – It is possible for the DBA to ‘offline’ or make unavailable, a rollback segment that has active transactions. It may be possible that active transactions are using this off lined rollback segment. In this case, the rollback is not really off lined; it is marked as ‘pending offline’. **SMON** will periodically try to ‘really’ offline it in the background until it can.

That should give you a flavor of what **SMON** does. The **SMON** process can accumulate quite a lot of CPU over time, and this should be considered normal. **SMON** periodically wakes up (or is woken up by the other backgrounds) to perform these housekeeping chores.

## RECO – Distributed Database Recovery

**RECO** has a very focused job; it recovers transactions that are left in a prepared state because of a crash or loss of connection during a two-phase commit (2PC). A 2PC is a distributed protocol that allows for a modification that affects many disparate databases to be committed atomically. It attempts to close the window for distributed failure as much as possible before committing. In a 2PC between N databases, one of the databases, typically (but not always) the one the client logged in to initially, will be the coordinator. This one site will ask the other N-1 sites if they are ready to commit. In effect, this one site will go to the N-1 sites, and ask them to be prepared to commit. Each of the N-1 sites reports back their ‘prepared state’ as YES or NO. If any one of the sites votes NO, the entire transaction is rolled back. If all sites vote YES, then the site coordinator broadcasts a message to make the commit permanent on each of the N-1 sites.

If after some site votes YES, they are prepared to commit, but before they get the directive from the coordinator to actually commit, the network fails or some other error occurs, the transaction becomes an *in-doubt distributed transaction*. The 2PC tries to limit the window of time in which this can occur, but cannot remove it. If we have a failure right then and there, the transaction will become the responsibility of **RECO**. **RECO** will try to contact the coordinator of the transaction to discover its outcome. Until it does that, the transaction will remain in its uncommitted state. When the transaction coordinator can be reached again, **RECO** will either commit the transaction or roll it back.

It should be noted that if the outage is to persist for an extended period of time, and you have some outstanding transactions, you can commit/roll them back manually yourself. You might want to do this since an in doubt distributed transaction can cause *writers to block readers* – this is the one time this can happen in Oracle. Your DBA could call the DBA of the other database and ask them to query up the status of those in-doubt transactions. Your DBA can then commit or roll them back, relieving **RECO** of this task.

## CKPT – Checkpoint Process

The checkpoint process doesn’t, as its name implies, do a checkpoint (that’s mostly the job of **DBWn**). It simply assists with the checkpointing process by updating the file headers of the data files. It used to be that **CKPT** was an optional process, but starting with version 8.0 of the database it is always started, so if you do a **ps** on UNIX, you’ll always see it there. The job of updating data files’ headers with checkpoint information used to belong to the **LGWR** (Log Writer) however, as the number of files increased along with the size of a database over time, this additional task for **LGWR** became too much of a burden. If **LGWR** had to update dozens, or hundreds, or even thousands of files, there would be a good chance sessions waiting to commit these transactions would have to wait far too long. **CKPT** removes this responsibility from **LGWR**.

## DBWn – Database Block Writer

The Database Block Writer (**DBWn**) is the background process responsible for writing dirty blocks to disk. **DBWn** will write dirty blocks from the buffer cache, usually in order to make more room in the cache (to free buffers for reads of other data), or to advance a checkpoint (to move forward the position in an online redo log file from which Oracle would have to start reading, in order to recover the instance in the event of failure). As we discussed previously, when Oracle switches log files, a checkpoint is signaled. Oracle needs to advance the checkpoint so that it no longer needs the online redo log file it just filled up. If it hasn’t



been able to do that by the time we need to reuse that redo log file, we get the ‘checkpoint not complete’ message and we must wait.

---

NOTE: Advancing logfiles is only one of many ways for checkpoint activity to occur. There are incremental checkpoints controlled by parameters such as `FAST_START_MTTR_TARGET` and other triggers that cause dirty blocks to be flushed to disk.

---

As you can see, the performance of `DBWn` can be crucial. If it does not write out blocks fast enough to free buffers (buffers that can be reused to cache some other block) for us, we will see waits of ‘Free Buffer Waits’ and ‘Write Complete Waits’ start to grow.

We can configure more than one `DBWn`, up to twenty in fact (`DBW0 ... DBW9, DBWa ... DBWj`). Most systems run with one database block writer but larger, multi-CPU systems can make use of more than one. This is generally done to distribute the workload of keeping a large block buffer cache in the SGA “clean”, flushing the dirtied (modified) blocks to disk.

Normally, the `DBWn` uses asynchronous I/O to write blocks to disk. With asynchronous I/O, `DBWn` gathers up a batch of blocks to be written, and gives them to the operating system. `DBWn` does not wait for the OS to actually write the blocks out, rather it goes back and collects the next batch to be written. As the OS completes the writes, it asynchronously notifies `DBWn` that it completed the write. This allows `DBWn` to work much faster than if it had to do everything serially. We’ll see later, in the *Slave Processes* section, how we can use I/O slaves to simulate asynchronous I/O on platforms or configurations that do not support it.

I would like to make one final point about `DBWn`. It will, almost by definition, write out blocks scattered all over disk – `DBWn` does lots of scattered writes. When you do an update, you’ll be modifying index blocks that are stored here and there and data blocks that are randomly distributed on disk as well. `LGWR`, on the other hand, does lots of sequential writes to the redo log. This is an important distinction, and one of the reasons that Oracle has a redo log and the `LGWR` process. Scattered writes are significantly slower than sequential writes. By having the SGA buffer dirty blocks and the `LGWR` process do large sequential writes that can recreate these dirty buffers, we achieve an increase in performance. The fact that `DBWn` does its slow job in the background while `LGWR` does its faster job while the user waits, gives us overall better performance. This is true even though Oracle may technically be doing more I/O than it needs to (writes to the log and to the datafile) – the writes to the online redo log could be skipped if, during a commit, Oracle physically wrote the modified blocks out to disk instead.

## LGWR – Log Writer

The `LGWR` process is responsible for flushing to disk the contents of the redo log buffer, located in the SGA. It does this:

- \* Every three seconds, or
- \* Whenever a commit is issued by any transaction, or
- \* When the redo log buffer is a third full or contains 1 MB of buffered data.

For these reasons, having an enormous (100’s of megabytes) redo log buffer is not practical – Oracle will never be able to use it all. The logs are written to with sequential

writes as compared to the scattered I/O **DBWn** must perform. Doing large batch writes like this is much more efficient than doing many scattered writes to various parts of a file. This is one of the main reasons for having a **LGWR** and redo logs in the first place. The efficiency in just writing out the changed bytes using sequential I/O outweighs the additional I/O incurred. Oracle could just write database blocks directly to disk when you commit but that would entail a lot of scattered I/O of full blocks – this would be significantly slower than letting **LGWR** write the changes out sequentially.

### **ARCn – Archive Process**

The job of the **ARCn** process is to copy an online redo log file to another location when **LGWR** fills it up. These archived redo log files can then be used to perform media recovery. Whereas online redo log is used to ‘fix’ the data files in the event of a power failure (when the instance is terminated), archive redo logs are used to ‘fix’ data files in the event of a hard disk failure. If you lose the disk drive containing the data file, `/d01/oradata/oral0g/system.dbf`, we can go to our backups from last week, restore that old copy of the file, and ask the database to apply all of the archived and online redo log generated since that backup took place. This will ‘catch up’ that file with the rest of the data files in our database, and we can continue processing with no loss of data.

**ARCn** typically copies online redo log files to at least two other locations (redundancy being a key to not losing data!). These other locations may be disks on the local machine or, more appropriately, at least one will be located on another machine altogether, in the event of a catastrophic failure. In many cases, these archived redo log files are copied off by some other process to some tertiary storage device, such as tape. They may also be sent to another machine to be applied to a ‘standby database’, a failover option offered by Oracle. We’ll discuss the processes involved in that below.

### **ASMB – Automatic Storage Management “B” process**

The **ASMB** process runs in a database instance that is making use of Automatic Storage Management (discussed in Chapter 3, Files). It is responsible to communicating to the ASM instance that is managing the storage and providing updated statistics to the ASM instance as well as provide a “heartbeat” to the ASM instance, letting it know that it is still alive and functioning.

### **RBAL - ReBALance**

The **RBAL** process runs in a database instance that is making use of Automatic Storage Management as well. It is responsible for processing a rebalance request (a redistribution request) as disks are added/removed to and from an ASM diskgroup.

### **Real Application Clusters (RAC) Processes**

RAC is a configuration of Oracle whereby multiple instances, each running on a separate node, typically a separate physical computer, in a cluster may mount and open a single database. It gives you the ability to have more than one instance accessing, in a full read write fashion, a single set of database files. The primary goals of RAC are twofold:

- \* **High Availability:** With Oracle RAC, if one node/computer in a cluster fails due to a software, hardware or human error – the other nodes may continue to function. The database will be accessible via the other nodes. You might have lost some computing power, but you haven't lost access to the database.
- \* **Scalability:** Instead of buying larger and larger machines to handle an increasing workload (known as vertical scalability), RAC allows you to add resources in the form of more machines in the cluster (known as horizontal scaling). Instead of trading your 4 CPU machine in for one that can grow to 8 or 16 CPUs, RAC gives you the option of adding another relatively inexpensive 4 CPU machine (or more than one)

The following processes are unique to a RAC environment, you will not see them otherwise.

#### **LMON – Lock Monitor Process**

The **LMON** process monitors all instances in a cluster to detect the failure of an instance. It then facilitates the recovery of the global locks held by the failed instance. It is also responsible for reconfiguring locks and other resources when instances leave or are added to the cluster (as they fail and come back online, or as new instances are added to the cluster in real time).

#### **LMD – Lock Manager Daemon**

The **LMD** process handles lock manager service requests for the global cache service (keeping the block buffers consistent between instances). It works primarily as a broker sending requests for resources to a queue that is handled by the **LMSn** processes. The **LMD** handles global deadlock detection/resolution and monitors for lock timeouts in the global environment.

#### **LMSn – Lock Manager Server Process**

This process is used exclusively in an Oracle Real Application Clusters (RAC) environment. RAC is a configuration of Oracle whereby more than one instance mounts and opens the same database. Each instance of Oracle in this case is running on a different machine in a cluster, and they all access in a read-write fashion the same exact set of database files.

In order to achieve this, the SGA block buffer caches must be kept consistent with respect to each other. This is one of the main goals of the LMSn processes. In earlier releases of OPS this was accomplished via a 'ping'. That is, if a node in the cluster needed a read consistent view of a block that was locked in exclusive mode by another node, the exchange of data was done via a disk flush (the block was pinged). This was a very expensive operation just to read data. Now, with the BSP, this exchange is done via very fast cache-to-cache exchange over the clusters high-speed connection.

You may have up to 10 LMSn processes per instance.

#### **LCK0 – Lock Process**

This process is very similar in functionality to the **LMD** described above, but handles requests for all global resources other than database block buffers.

#### **DIAG – Diagnosability Daemon**

The **DIAG** process is used exclusively in a RAC environment. It is responsible to monitoring the overall “health” of the instance and captures information needed in the processing of instance failures.

## Utility Background Processes

These background processes are totally optional, based on your need for them. They provide facilities not necessary to run the database day-to-day, unless you are using them yourself, or are making use of a feature that uses them, such as the new Oracle 10g diagnostic capabilities..

These processes will be visible in UNIX as any other background process would be – if you do a **ps** you will see them. In my **ps** listing from above and reproduced in part below, you can see that I am have

- \* Job queues configured – the CJQ0 process is the job queue coordinator,
- \* Oracle Advanced Queues (AQ) is configured evidenced by the Q000 (AQ queue process) and QMNC (AQ monitor process)
- \* Automatic SGA sizing, as evidenced by the MMAN (Memory Manager) process.
- \* The Oracle 10g manageability/diagnostic features enabled, evidenced by the MMON (Manageability Monitor) and MMNL (Manageability Monitor Light) processes.

```
ora10g 5894 1 0 16:17 ? 00:00:00 ora_mman_ora10g
ora10g 5906 1 0 16:17 ? 00:00:00 ora_cjq0_ora10g
ora10g 5920 1 0 16:17 ? 00:00:00 ora_qmnc_ora10g
ora10g 5922 1 0 16:17 ? 00:00:00 ora_mmon_ora10g
ora10g 5924 1 0 16:17 ? 00:00:00 ora_mmnl_ora10g
ora10g 5939 1 0 16:28 ? 00:00:00 ora_q000_ora10g
```

Let’s take a look at the various processes you might see depending on the features you are using

## CJQ0 and Jnnn Processes – (Job Queues)

In the first 7.0 release, Oracle provided replication. This was done in the form of a database object known as a *snapshot*. The internal mechanism for refreshing, or making current, these snapshots was the job queues. This was a process that monitored a job table that told it when it needed to refresh various snapshots in the system. In Oracle 7.1, Oracle Corporation exposed this facility for all to use via a database package called **DBMS\_JOB**. What was solely the domain of the snapshot in 7.0 become the ‘job queue’ in 7.1 and later versions. Over time, the parameters for controlling the behavior of the queue (how frequently it should be checked and how many queue processes there should be) changed their name from **SNAPSHOT\_REFRESH\_INTERVAL** and **SNAPSHOT\_REFRESH\_PROCESSES** to **JOB\_QUEUE\_INTERVAL** and **JOB\_QUEUE\_PROCESSES** and in current releases, only the **JOB\_QUEUE\_PROCESSES** parameter is exposed as a user tunable setting.

You may have up to 1000 job queue processes. Their names will be **J000**, **J001**, ... , **J999**. These job queues’ processes are used heavily in replication as part of the materialized view refresh process. Streams based replication (new with Oracle 9i Release 2) utilizes AQ for replication. Developers also frequently use them in order to schedule one-off (background)

jobs or recurring jobs. For example, to send an email in the background, or process a long running batch process in the background. By doing some work in the background, you can make a long task seem to take much less time to the impatient end user (they feel like it went faster, even though it might not be done yet), this is similar to what Oracle does with **LGWR** and **DBWn** processes, rather than waiting for them to complete all tasks in real time, they do much of their work in the background.

The **J000** processes are very much like a shared server, but with aspects of a dedicated server. They are shared – they process one job after the other, but they manage memory more like a dedicated server would (UGA in the PGA issue). Each job queue process will run exactly one job at a time, one after the other, to completion. That is why we may need multiple processes if we wish to run jobs at the same time. There is no threading or pre-empting of a job. Once it is running, it will run to completion (or failure). You will notice that the **Jnnn** processes come and go over time, that is, if you configure up to 1,000 of them, you will not see 1,000 of them start up with the database. Rather a sole process – the **CJQ0** (the Job Queue Coordinator) will start up and as it sees jobs that need to be run in the job queue table, it will start the **Jnnn** processes. As the **Jnnn** processes complete their work and discover no new jobs to process, they will start to exit – to go away. So, if you schedule most of your jobs to run at 2am in the morning, when no one is around, you might well never see the **Jnnn** processes as they would come and go during the night and since they are not needed during the day, they just wouldn't be there.

### **QMNC and Qnnn – Advanced Queues**

The **QMNC** process is to the AQ tables what the **CJQ0** process is to the job table. It monitors the Advanced Queues and alert waiting 'dequeuers' of messages, that a message has become available. They are also responsible for queue propagation – the ability of a message enqueued (added) in one database to be moved to a queue in another database for dequeuing. The **Qnnn** process are to the **QMNC** process what the **Jnnn** processes were to the **CJQ0** process. They are notified by the **QMNC** process of work that needs to be performed and process the work.

The **QMNC** and **Qnnn** processes are optional background process. The parameter **AQ\_TM\_PROCESSES** specifies creation of up to ten of these processes named **Q000**, ... , **Q009** and a single **QMNC** process. If **AQ\_TM\_PROCESSES** is set to 0, there will be no **QMNC** or **Qnnn** processes. Unlike the **Jnnn** processes used by the job queues above, the **Qnnn** processes are persistent, if you set **AQ\_TM\_PROCESSES** to 10, you will see 10 **Qnnn** process and the **QMNC** process at database startup and for the entire life of the instance.

### **EMNn – Event Monitor Processes**

The **EMNn** is part of the Advanced Queue architecture. It is a process that is used to notify queue subscribers of messages they would be interested in. This notification is performed asynchronously. There are Oracle Call Interface (OCI) functions available to register a callback for message notification. The callback is a function in the OCI program that will be invoked automatically whenever a message of interest is available in the queue. The **EMNn** background process is used to notify the subscriber. The **EMNn** process is started automatically when the first notification is issued for the instance. The application may then issue an explicit **message\_receive(dequeue)** to retrieve the message.

## **MMAN – Memory Manager**

This process is new with Oracle 10g and is used by the Automatic SGA sizing feature. The **MMAN** process coordinates the sizing and resizing of the shared memory components (the default buffer pool, the shared pool, the java pool and the large pool).

## **MMON, MMNL and Mnnn – Manageability Monitors**

These processes are used to populate the AWR (Automatic Workload Repository) new in Oracle 10g. The MMNL (Manageability Monitor Light) flushes statistics from the SGA to database tables on a scheduled basis. The MMON (Manageability Monitor) process is used to “auto” detect database performance issues and implement the new self tuning features. The Mnnn processes are similar to the Jnnn or Qnnn processes for the job queues, the MMON process will request these slave processes to perform work on it’s behalf. The Mnnn processes are transient in nature, they will come and go as needed.

## **CTWR – Change Tracking Processes**

This is a new optional process of the Oracle 10g database. The Change Tracking Process is responsible for maintaining the new change tracking file described in Chapter 3, Files.

## **RVWR – Recovery Writer**

This process, another new optional process of the Oracle 10g database, is responsible for maintaining the before images of blocks in the flash recovery area (described in Chapter 3, Files) used with the **FLASHBACK DATABASE** command.

## **Remaining Utility Background Processes**

So, is that the complete list? No, there are others. For example – Oracle Data Guard has a set of processes associated with it to facilitate the shipping of redo information from one database to another and apply it (see the Data Guard Concepts and Administration Guide from Oracle for details). There are processes associated with the new Oracle 10g Datapump utility that you will see during certain Datapump operations. There are Streams apply and capture processes as well. The above list however covers most of the common background processes you will see day to day.

## **Slave Processes**

Now we are ready to look at the last class of Oracle processes, the ‘slave’ processes. There are two types of slave processes with Oracle – I/O slaves and Parallel Query slaves.

### **I/O Slaves**

I/O slaves are used to emulate asynchronous I/O for systems, or devices, that do not support it. For example, tape devices (notoriously slow) do not support asynchronous I/O. By utilizing I/O slaves, we can mimic for tape drives what the OS normally provides for disk drives. Just as with true asynchronous I/O, the process writing to the device, batches up a large amount of data and hands it off to be written. When it is successfully written, the writer

(our I/O slave this time, *not* the OS) signals the original invoker, who removes this batch of data from their list of data that needs to be written. In this fashion, we can achieve a much higher throughput, since the I/O slaves are the ones spent waiting for the slow device, while their caller is off doing other important work getting the data together for the next write.

I/O slaves are used in a couple of places in Oracle –**DBWn** and **LGWR** can make use of them to simulate asynchronous I/O and the RMAN (**R**ecovery **MAN**ager) will make use of them when writing to tape.

There are two parameters controlling the use of I/O slaves:

- \* **BACKUP\_TAPE\_IO\_SLAVES** – This specifies whether I/O slaves are used by the RMAN to backup, copy, or restore data to tape. Since this is designed for *tape* devices, and tape devices may be accessed by only one process at any time, this parameter is a Boolean, not a number of slaves to use as you would expect. RMAN will start up as many slaves as is necessary for the number of physical devices being used. When **BACKUP\_TAPE\_IO\_SLAVES = TRUE**, an I/O slave process is used to write to, or read from a tape device. If this parameter is **FALSE** (the default), then I/O slaves are not used for backups. Instead, the shadow process engaged in the backup will access the tape device.
- \* **DBWR\_IO\_SLAVES** – This specifies the number of I/O slaves used by the DBWn process. The DBWn process and its slaves always perform the writing to disk of dirty blocks in the buffer cache. By default, the value is 0 and I/O slaves are not used.

These I/O slaves will appear with the name Innn, where nnn will be a number.

## Parallel Query Slaves

Oracle 7.1 introduced parallel query capabilities into the database. This is the ability to take a SQL statement such as a **SELECT**, **CREATE TABLE**, **CREATE INDEX**, **UPDATE**, and so on and create an execution plan that consists of *many* execution plans that can be done simultaneously. The outputs of each of these plans are merged together into one larger result. The goal is to do an operation in a fraction of the time it would take if you did it serially. For example, if you have a really large table spread across ten different files, 16 CPUs at your disposal, and you needed to execute an ad-hoc query on this table, it might be advantageous to break the query plan into 32 little pieces, and really make use of that machine. This is as opposed to just using one process to read and process all of that data serially.

## Summary

That's it – the three pieces to Oracle in the last three chapter. We've covered the files used by Oracle, from the lowly, but important parameter file, to data files, redo log files and so on. We've taken a look inside the memory structures used by Oracle, both in the server processes and the SGA. We've seen how different server configurations such as shared server versus dedicated server mode for connections will have a dramatic impact on how memory is used by the system. Lastly we looked at the processes (or threads depending on the operating system) that make Oracle do what it does. Now we are ready to look at the implementation of some other features in Oracle such as Locking, Concurrency Controls and Transactions in the following chapters.