Waiting and Blocking Issues

CHAPTER 1

IN THIS CHAPTER

- Wait Types
- Troubleshooting Blocking
- Identifying Blocking
- Identifying the Cause of Blocking
- Resource Type Specifics
- Deadlocks
- Monitoring Blocking
- Conclusion
- Other Resources

By Santeri Voutilainen

I will start this chapter on blocking by talking about waiting. Why would I start a chapter on blocking with waiting? Well, the two are very much related, and they are often treated as synonyms. Because they are so related, the SQL Server concepts and tools related to each are intermingled; therefore, it is important to distinguish one from the other.

Conceptually, *waiting* usually refers to an idle state in which a task is waiting for something to occur before continuing execution. This "something" might be the acquisition of a synchronization resource, the arrival of a new command batch, or some other event. Although this description is generally accurate, there is an important caveat: Not all tasks identified as waiting within SQL Server are in fact idle. This is because the waiting classification is sometimes used to indicate that the task is executing a particular type of code. This code is often outside the direct control of SQL Server. In effect, the SQL Server task is waiting for the completion of the external code.

Wait Types

All waits within SQL Server are categorized into wait types. The current wait type for a task is set based on the reason for the wait. Each wait type is given a name that describes, to some extent, the location, component, resource, or reason for the wait.

Although some of the names can be somewhat cryptic (I cover some of these later), others are self-explanatory. The list of wait types is available from the `sys.dm_os_wait_stats` dynamic management view (DMV). By default, the output from this DMV is not the full list. For SQL Server 2005, the SQL Server product team opted not to include some wait types that fall under one of the following three categories:

- Wait types that are never used in SQL Server 2005; note that some wait types not excluded are also never used.
- Wait types that can occur only at times when they do not affect user activity, such as during initial server startup and shutdown, and are not visible to users.
- Wait types that are innocuous but have caused concern among users because of their high occurrence or duration.

Unfortunately, the omission of these wait types can also lead to concern because the last category of excluded wait types does appear in other sources of waits—namely, the `sys.dm_os_waiting_tasks` DMV. The complete list of wait types is available by enabling trace flag 8001. The only effect of this trace flag is to force `sys.dm_os_wait_stats` to display all wait types.

```
dbcc traceon (8001, -1)
```

NOTE

This trace flag is undocumented, and, like all undocumented features, it is unsupported, so you use it at your own risk.

The list of wait types can be divided into four basic categories: Resource, Timer/Queue, IO, and External. The Resource waits category is by far the largest. It covers waits for resources such as synchronization objects, events, and CPU. The Timer/Queue wait category includes waits where the task is waiting for the expiration of a timer before proceeding or when a task is waiting for new items in a queue to process. The IO category contains most wait types related to IO operations. Both network and disk IO are included. The External waits category covers the cases mentioned earlier, where the task is executing certain types of code, often external to SQL Server, such as extended stored procedures.

The list of tasks currently in the waiting state is available from `sys.dm_os_waiting_tasks`. With regard to waiting, this DMV includes information identifying the waiting tasks, the duration of the wait, the wait type, and, in some cases, additional information about what is being waited for. The DMV also includes blocking-specific information—namely, the identity of the process blocking the continued execution of the task, when the identity is known. It is also the root source of blocking information, so knowing how to distinguish blocking from plain waiting using this DMV is important.

Blocking is distinguished from waiting in that the wait is not voluntary; instead, it is forced on the waiting worker by another task preventing worker's task from proceeding. This occurs when the tasks attempt simultaneous access of a shared resource.[1] This definition is generally considered to exclude the wait types in the External and Timer/Queue categories from indicating that the waiting task is blocking, even though the definition does not strictly exclude the External category. Nevertheless, short sections on both External and Timer/Queue waits are included toward the end of the chapter.

Troubleshooting Blocking

There are three general steps to investigate and deal with blocking:

1. Identify that blocking occurred.
2. Identify the cause of the blocking.
3. Eliminate the cause of the blocking.

This chapter focuses mostly on the first two steps. Because blocking can occur in a variety of components, and the causes can be specific to those components, the solutions often are component-specific, too. Rather than include detailed information on these components in this chapter, I make references to the documentation for those components. However, I have included causes and solutions for some of the more common blocking types.

Note that many of the examples and the discussion assume SQL Server 2005 SP1 (or later). This is particularly the case for the `sys.dm_os_waiting_tasks` DMV. Several changes and fixes for this DMV were included in SP1. The SQL Server 2005 RTM behavior would have significant differences compared to what is described next.

Identifying Blocking

Identifying Blocking Using *sys.dm_os_waiting_tasks*

In SQL Server 2005, the `sys.dm_os_waiting_tasks` DMV is the fundamental repository for blocking information within the server. Before discussing how it can be used to detect the existence of blocking, we cover some of its more important columns.

`sys.dm_os_waiting_tasks` contains three groups of columns: those that identify the waiter, those that identify the blocker (if applicable), and those that provide information about the wait.

---

1 In general, this requires the existence of two tasks. However, it is possible for a task to block itself. In reality, this occurs rarely and generally results in a deadlock. Its existence can affect scripts used to monitor or investigate blocking.

The columns that identify the waiting task are the `waiting_task_address`, `session_id`, and `exec_context_id` columns. The `waiting_task_address` contains the internal memory address of the object that represents the task. This uniquely identifies a task within SQL Server. The `session_id` represents the user session with which the task is currently associated. The association of a task with a session lasts only for the duration of the task, generally a batch. The blocking task is identified by the `blocking_task_address`, `blocking_session_id`, and `blocking_exec_context_id`. These have the same semantics as the columns identifying the waiting task.

The relationship between the task address and `session id` columns should be noted. Only sessions that are currently executing a statement within SQL Server have an associated task. However, a task need not be associated with a session ID: this occurs with various system tasks. Because all waiting in SQL Server is done by tasks, it is not possible for a session that is not currently associated with a task to be waiting or blocked. Therefore, the `waiting_task_address` is never null, but the `session_id` and `exec_context_id` columns may be null if the task is not associated with a session. On the flip side, a task can be blocked by either an active task or an inactive session. This occurs if the resource involved in the wait can be held by a session across tasks (that is, across batches). In SQL Server 2005, the only type of resource that can cause blocking and that is held across batches are locks. Note that although some other resources such as memory associated with a session are held across batches, these do not cause direct blocking but are rather represented through proxy waits such as the `LOWFAIL_MEMMGR_QUEUE` wait type. Locks are held by transactions, and because it is possible for a transaction to contain multiple batches, it is thus possible for locks to be held by a session that has a transaction active while performing operations on the client side and thus would not have an associated task within the server.

The identity of the blocking task is not always known. In these cases, the columns identifying the blocking task are null. This is rather common because blocker information is not available for most wait types—either because it does not make sense for the particular wait type or because the information is not tracked (due to, for example, performance concerns).

As its name implies, the `wait_type` column contains the current wait type for the waiting task. Similarly, the `wait_duration_ms` column contains the duration of the current wait. The `resource_address` column provides the memory address of the resource on which the task is waiting. This is mainly useful as an identifier to differentiate blocking on different instances of a resource when the wait type names are the same. The `resource_description` column can also provide differentiating information in such cases, but it is only populated with useful information by a handful of wait types—all lock and latch wait types, the `CXPACKET` and `THREADPOOL` wait types.

The simplest method for detecting blocking using `sys.dm_os_waiting_tasks` is simply to run the T-SQL query `select * from sys.dm_os_waiting_tasks` and treat any row in the output as signifying blocking. This, however, is not very useful, because there will almost always be at least a handful of task wait states from the categories that can generally be excluded from blocking investigations. For example, the deadlock monitor thread usually

is listed as waiting with a wait type of REQUEST_FOR_DEADLOCK_SEARCH, which merely indicates that it is pausing between deadlock detection events and is not blocked by anything. The one time this category of waits should not be ignored is if the tasks in these waits are blocking others, which is almost never the case except with the WAITFOR wait type.

Although it is possible to memorize the set of excludable wait types and mentally exclude them from the result set, this is generally not efficient. A view built on top of sys.dm_os_waiting_tasks can help.

The amalgam.innocuous_wait_types view available on the accompanying CD produces a list of innocuous wait types that usually can be ignored. The amalgam.dm_os_waiting_tasks_filtered view uses the helper view to filter out these innocuous waits:

```
create view amalgam.dm_os_waiting_tasks_filtered
as
select *
from sys.dm_os_waiting_tasks
where wait_type not in (
      select *
      from amalgam.innocuous_wait_types)
go
```

Any rows left in the filtered rowset represent tasks that are truly blocked. This set can be further analyzed and narrowed based on the severity, cause, and nature of the blocking. You learn more about this analysis in the sections dealing with identifying the cause of blocking.

Why Not Use *sysprocesses* or *sys.dm_exec_requests*?

In previous versions of SQL Server, the DMV of choice for investigating waiting and blocking was sysprocesses. Although this DMV exists in SQL Server 2005, it has been deprecated and replaced by a set of new DMVs. The new DMV that contains the waiting and blocking information from sysprocesses is sys.dm_exec_requests. Although both of these DMVs contain information that you can use to investigate blocking and waiting, there are good reasons not to use them.

The main reason you should not use these views is that they do not provide the level of detailed information that is available from sys.dm_os_waiting_tasks. Sysprocesses and sys.dm_exec_requests display session-level information. This means that they do not contain system processes that are not associated with session IDs. Their session basis also makes it harder to handle parallel queries where multiple tasks are executing under the same session ID. sys.dm_exec_requests displays just one entry for each session. Because of this, sys.dm_exec_requests is not usable when investigating blocking involving parallel queries, because it displays only the blocked/waiting status of the parent task, not the child tasks.

The session-level focus also affects the display of blocker information. Both `sysprocesses` and `sys.dm_os_waiting_tasks` display only the session ID of the blocking task rather than the specific task in case of parallel queries.

Finally, the resource descriptions for `sysprocesses` and `sys.dm_exec_requests` are not as complete as those in `sys.dm_os_waiting_tasks`, which provides the most complete set of resource descriptions available. Further, the `resource_address` column in `sys.dm_os_waiting_tasks` can be used to differentiate between multiple resources that do not have resource descriptions. With the other two DMVs, these multiple resources could not be distinguished from each other.

Although `sys.dm_exec_requests` should not be used as the primary source of blocking information, it does contain information that is useful when you're investigating certain types of blocking, so it should not be ignored.

Statistically Identifying Blocking

`Sys.dm_os_waiting_task`, although a useful resource, is not the only resource for detecting blocking. Statistical information on waiting and blocking is available from several sources. This information can at times prove more useful than current wait and blocking data from `sys.dm_os_waiting_tasks` because the statistical information can differentiate between occasional and frequent, and short and long waits. Frequent long blocking is generally of much more concern than occasional short blocking.

sys.dm_os_wait_stats

In addition to providing a list of wait types, as mentioned in the first section, `sys.dm_os_wait_stats` provides statistics for each wait type. The information provided includes the number of times a wait with a given wait type has occurred, the total duration of those waits, and the maximum wait time for a single wait.

A single query over this DMV can be used to identify wait types with long average waits. Keep in mind that no significance should be assigned to the absolute wait counts returned by a single snapshot of the view, because these counts are cumulative since the last reset, which might have occurred some time ago. The wait counts are interesting when measured over a known time span. The deltas between two snapshots indicate the severity of blocking during that interval. These snapshots can be generated manually by running a query over the DMV twice and calculating the difference, or, more conveniently, using the SQLDiag tool that ships with SQL Server 2005.

```
-- Create temporary tables to store the initial
-- and final snapshots
--
create table #StatsInitial (
      wait_type sysname,
      waiting_tasks_count bigint,
      wait_time_ms bigint,
      signal_wait_time_ms bigint);
create table #StatsFinal (
            wait_type sysname,
            waiting_tasks_count bigint,
```

```
      wait_time_ms bigint,
      signal_wait_time_ms bigint);
-- Create indexes for join performance
--
create index idxInitialWaitType
      on #StatsInitial (wait_type);
create index idxFinalWaitType
      on #StatsFinal (wait_type);

-- Create an initial snapshot
--
insert into #StatsInitial
select wait_type,
      waiting_tasks_count,
      wait_time_ms,
      signal_wait_time_ms
from amalgam.dm_os_wait_stats_filtered;

 -- Wait for a ten second delay
 -- This delay can be adjusted to suit your needs
 -- and preferences
 --
 waitfor delay '00:00:10';

 -- Create the final snapshot
 --
 insert into #StatsFinal
 select wait_type,
      waiting_tasks_count,
      wait_time_ms,
      signal_wait_time_ms
 from amalgam.dm_os_wait_stats_filtered;

 -- Report any wait types that had waits during
 -- the wait delay, and the number and duration
 -- of the waits
 --
 select f.wait_type,
      f.waiting_tasks_count - i.waiting_tasks_count
            as wait_tasks_count_delta,
      f.wait_time_ms - i.wait_time_ms
            as wait_time_ms_delta,
      f.signal_wait_time_ms - i.signal_wait_time_ms
            as signal_wait_time_delta
      from #StatsFinal f join #StatsInitial i
      on f.wait_type = i.wait_type
 where f.waiting_tasks_count - i.waiting_tasks_count > 0
order by f.waiting_tasks_count - i.waiting_tasks_count
desc;

-- Finally drop the tables
--
drop table #StatsInitial;
drop table #StatsFinal;
go
```

Note that, as written, the preceding queries exclude the innocuous wait types by referencing the `amalgam.dm_os_wait_stats_filtered` view.

Note that although it is possible to simplify this process by clearing the stats and then running the query, this approach has several drawbacks, including the loss of historical statistics, and the negative effect it can have on other tools that are also using the DMV for wait statistics, including some ISV monitoring tools:

```
dbcc sqlperf ('sys.dm_os_wait_stats', CLEAR)
waitfor delay '00:00:10';
select *
from amalgam.dm_os_wait_stats_filtered;
```

When determining the severity of blocking based on these statistics, it is important to keep in mind that these are aggregated statistics for all waits of each type. For many waits, this might not point to a single resource. Prime examples of this are latch and lock waits. The page latch count waits cover all pages in all databases, files, and objects. The same is true for lock waits.

Similarly, comparing overall wait counts is not as important as the rate of waits (waits per second) or the average duration of the waits. A few long waits generally indicate a localized issue, whereas many short or medium-length waits are more indicative of hot spotting or throughput bottlenecks.

Performance Counters

You can also use performance monitor counters to detect blocking. Some of the information provided by the performance counters mirrors the information available from `sys.dm_os_wait_stats`, but other counters provide information data that is available only from performance monitor counters.

Although the easiest way of monitoring SQL Server performance counters is via the SQLDiag tool, SQL Server's counters are also available in rowset form through the `sys.dm_os_performance_counters` DMV. The data in this DMV is the raw data reflected in the performance counters, so some of it requires some processing in order to be useful.

The Wait Statistics group accumulates counters for various types of waits. The Process blocked counter in the General Statistics group is a useful one that you can use to track the number of blocked processes without needing to query `sys.dm_os_waiting_tasks`.

Notification-Based Detection

SQL Server 2005 includes a built-in proactive method of blocking notification. This is the Blocked Process Report event, which can be captured in a trace and that can be used to trigger code to deal with the blocking or notify operators. This event is triggered when a wait exceeds the specified threshold. The usefulness of this event for monitoring general blocking is somewhat limited by the fact that it only detects blocking on resources that support deadlock detection. Resources that support deadlock detection are listed in Table 1-1 along with the corresponding wait types.

TABLE 1-1 Resources that support deadlock detection

Deadlock-Detectable Resource	Corresponding Wait Types
Locks	LCK_M_*
Worker threads	Any deadlock detectable resource wait types, but generally lock waits
Memory	RESOURCE_SEMAPHORE
Parallel query execution	CXPACKET, EXCHANGE, EXECSYNC
	Multiple active result set (MARS) resources TRANSACTION_MUTEX, MSQL_DQ, MSQL_XP, ASYNC_NETWORK_IO
CLR resources	SQLCLR_APPDOMAIN, CLR_MONITOR, CLR_RWLOCK_READER, CLR_RWLOCK_WRITER

The blocking threshold trigger is set using the `'blocked process threshold'` option of the `sp_configure` command. Note that this is an advanced option, so the `'show advanced options'` option must first be enabled. When the specified blocking threshold duration is crossed, the event is produced. As with other events, you can configure this event to produce an alert or run diagnostic scripts:

```
sp_configure 'show advanced options', 1;
go
reconfigure;
go
sp_configure 'blocked process threshold',
      <threshold-in-seconds>;
go
reconfigure;
```

The event includes a blocking graph similar to that produced for deadlocks. When the event fires because of blocking on a lock resource, identifying information for the lock resource is included in the event columns.

Using the Blocked Process Threshold event is particularly useful to capture unexpectedly long waits on lock and MARS resources. The threshold value should be established after having achieved an acceptable level of blocking by tuning the application and SQL Server. Under these conditions, this event provides a lightweight blocking monitoring mechanism for these types of blocking.

Identifying the Cause of Blocking

After blocking has been discovered, the next step is to identify the cause. The root cause is often specific to a particular wait type, but several shared concepts and operations help determine the root cause.

Current Statements and Plans

Although the name of the wait type and the wait description can help identify the component and resource in which blocking is occurring, the current statements for both the blocked and blocking tasks are useful to have for gathering more information. The current statement for both sessions is available from the `sys.dm_exec_requests` DMV. This works only for sessions that are currently executing. It is not possible, using DMVs, to find the text of the last batch for a session, although you can use `DBCC INPUTBUFFER` in many circumstances. `sys.dm_exec_sessions` does, however, contain the time the last batch was issued. You can use this to determine a minimum duration since the earliest possible start of blocking. Some blocking, especially lock-based blocking, is often affected by the query plans chosen. The current query plan is also available from `sys.dm_exec_requests`. The `amalgam.current_statements_and_plans` view included on the CD accompanying this book hides some of the messy details.

For monitoring purposes, it is useful to note that the source of the current statement text, `sys.dm_exec_requests.sql_handle`, is a durable handle within an instance of SQL Server. This means that the same `sql_handle` value is always used for the same batch on a particular instance of SQL Server. Note, however, that to retrieve the actual text corresponding to the `sql_handle`, the batch must be in cache. For dynamic SQL, the `sql_handle` can even be compared across SQL Server instances. For objects, such as stored procedures and functions, the `sql_handle` is derived from the database and object IDs, so it varies across instances.

A word of caution is in order regarding the current statement and query plan for the blocking tasks. A blocking task's current statement may not be the actual cause of the blocking. Given that some locks are held longer than a single statement, this is particularly the case with locks where the lock that is causing the blocking could have been acquired by any statement in the blocking task's transaction, not just the current statement.

Blocking Patterns

The output of `sys.dm_os_waiting_tasks` may exhibit several blocking patterns. These are relatively easy to identity, and their classification can be useful in determining which blocking issues to investigate first.

The first pattern is that of a single long wait. This is characterized by one task (or a small number of tasks) waiting on a unique wait type and resource for a long time. This type of blocking might not have a significant effect on the server's throughput, but it can dramatically increase the response time for the blocked query. Externally the response time delay could have more-significant effects. The following query lists waits that have lasted longer than ten seconds. The threshold level is relative and should be adjusted to the tolerances of each individual system. You can further adjust it to specific levels based on the wait type or some other qualifier:

```
select *
from amalgam.dm_os_waiting_tasks_filtered
```

```
-- adjust the following threshold as appropriate
where wait_duration_ms > 10000
order by wait_duration_ms desc
```

The next pattern is a large number of direct waits for a single resource. There is no extended blocking chain in this case—each task is blocked by the same single task. This blocking may start having an effect on the server throughput, because fewer workers are available to process the remainder of the workload. Response time is also affected for all queries. This pattern is often a hot spot, and because all contention is on a single resource, this type of blocking is generally relatively easy to resolve.

The third pattern is a large number of single waits on different resources. These can be much harder to handle than the previous categories, because there is no clear target to investigate. One option is to do short investigations of each of these waits. This might uncover a common trait between them. When this category is coupled with short wait times, it may be better to use statistical approaches to determining the cause of the blocking.

The final pattern is a blocking chain with multiple levels of blocking. Each level may have different types of waits as well as categories of blocking. This pattern is effectively a combination of the preceding three categories, and, as such, each level of blocking can be investigated separately.

Blocking Chains

As mentioned earlier, not all blocking is created equal. Some forms of blocking affect throughput or response times more than others. One measure of this is the number of other tasks blocked by a given blocking task. Some of these tasks are directly blocked by the head blockers; others are blocked indirectly. Indirect blocking is blocking where task T2 is blocked by T1 and T3 is blocked by T2, so T3 is indirectly blocked by T1 because it cannot proceed until T2 can proceed, which cannot occur until T1 unblocks T2.

Finding head blockers is deceptively simple: Just find all blocking tasks that are not blocked themselves. The task is made slightly more complex by the fact that not all waits have blocking information. Thus, if task T2 is blocked by T1, which itself is waiting but has no identified blocker, who should be marked as the head blocker? In order not to lose sight of this type of blocking chain, it is useful to consider tasks that are blocking others but do not have an identified blocker themselves as head blockers. This is especially the case for voluntary waits such as `WAITFOR` queries where there truly is no blocker:

```
create view amalgam.head_blockers
as
select blocking_task_address
as head_blocker_task_address,
          blocking_session_id
as head_blocker_session_id
from sys.dm_os_waiting_tasks
where blocking_task_address is not null OR
        blocking_session_id is not null
except
select waiting_task_address, session_id
```

```
from sys.dm_os_waiting_tasks
where blocking_task_address is not null OR
       blocking_session_id is not null
go
```

Note two important aspects about this query. First, it qualifies a task/session as a head blocker if it is blocking some task and it itself does not have an identified blocker—in fact, it may not even exist in the DMV as a waiter. Second, not all waiting tasks without an identified blocker are head blockers. This is the case for wait types that do not provide the ability to determine the blocker, in which case the head blocker may be waiting for a resource, but the identity of the blocker cannot be determined.

You can use the list of head blockers to calculate the number of direct and indirect blockers. This measure is one factor to consider when deciding which blocking chains to tackle first. You can use the `amalgam.blocking_chain` view to calculate the direct blocking counts for any blocker, and indirect blocking counts for head blockers. The view uses a recursive common table expression (CTE). The indirect count relies on the fact that at every level of the blocking chain, the head blocker information is maintained. Because of the possibility that any given snapshot of `sys.dm_os_waiting_tasks` may contain blocking chain cycles, the `maxrecursion` option should always be specified. This option could not be included as part of the view because option clauses are not allowed in views. Blocking chain cycles may exist due to three reasons:

- A deadlock exists, but the deadlock monitor has not yet detected it.
- A deadlock exists, but it cannot be detected, because it involves resources that do not participate in deadlock detection but populate the blocking information. This is the case, for example, with latches.
- No deadlock exists, but it appears as if one does exist (due to timing conditions when `sys.dm_os_waiting_tasks` was materialized).

```
-- Count of directly blocked tasks per blocker
--
select blocking_task_address,
     blocking_session_id,
     count(*) as directly_blocked_tasks
from amalgam.blocking_chain
group by blocking_task_address, blocking_session_id
option (maxrecursion 128)

-- Count of indirectly blocked tasks for each
-- head blocker
--
select head_blocker_task_address,
     head_blocker_session_id,
     count(*) as indirectly_blocked_tasks
from amalgam.blocking_chain
group by head_blocker_task_address,
     head_blocker_session_id
option (maxrecursion 128)
```

Resource Type Specifics

Beyond the common tools covered previously, much of the details of determining the cause of blocking and resolving it are specific to each wait type. Next, we'll cover some of the more common wait types.

Latches

Latches are short-term synchronization objects. Although originally used mainly for synchronization of physical access to database pages, their use in SQL Server as a general synchronization primitive has become widespread. SQL Server 2005 has more than 120 distinct usages of latches. Certain types of latches are a common source of blocking and, unfortunately, latch blocking is often hard to investigate and resolve (because of the scarcity of diagnostic information available for them). Fortunately, SQL Server 2005 provides much more information than was available in previous releases.

Latch waits are divided into two main groups: page latch waits and nonpage latch waits. Each of these groups can be subdivided into two subgroups. The main groups are the `PAGELATCH` and `PAGEIOLATCH`, and `TRANMARKLATCH` and `LATCH` wait base wait types, often referred to as page and nonpage latches, respectively. The `TRANMARKLATCH` group can be treated as any other nonpage latch even though it has the special status of having its own wait types.

In addition to these groups, different wait types exist for the latch mode being requested. The modes for each of these are `NL`, `KP`, `SH`, `UP`, `EX`, and `DT`. (Lock modes are defined in the Books Online topic `sys.dm_os_wait_stats`.) The actual wait type is formed by appending one of the modes to the group name (for example, `LATCH_EX`). Of the six modes, three are much more common the others. Waits for the `NL`, `KP`, and `DT` modes are rarely, if ever, seen. The `NL` mode is in fact never used. Although `KP` use is common, it only conflicts with `DT`, which is rarely used, so waits of either are quite rare.

Blocking Information

Blocking task information for latch waits is provided under certain circumstances. This information is not available for all latch waits because latches track information for only certain types of owners so as to remain lightweight. The blocking task information is known when a latch is held in `UP`, `EX`, or `DT` modes. The common factor with these modes is that a latch can be held in only one of these modes at a time and by only a single task, whereas `KP` and `SH` latches can be held by multiple tasks simultaneously. It is important to note that the available blocking information is not a factor of the mode specified in the wait type but rather a factor of the mode in which the latch is held. The mode specified in the wait type indicates the requested mode, not the blocking mode. If a latch is held in one of the preceding modes, all waiters for that latch are marked as blocked by the task that owns the latch in one of the preceding modes. Note that the blocking task information may change during a single uninterrupted wait. An example of this is the case in which the latch is held in both `SH` and `UP` modes and a task requests it in `EX` mode.

Although both the SH and UP modes are held by their respective tasks, the EX request is reported as blocked by the owner of the UP mode. When the UP mode is released while the SH is still held, the blocking information for EX reverts to unknown, because the owner of the SH mode is not available.

Grant Order

There are a few important aspects to the latch grant order. For the most part, latches are granted in first-in-first-out (FIFO) order, and any new requests need to wait if there are any other waiters—even if the requested mode is compatible with the granted modes. This is done to avoid starvation of the waiting task. Two exceptions apply to these rules. The first is that KP requests never need to wait to be granted unless the current mode is DT. The second is that when granting waiting requests after a release of a latch, all compatible requests are granted, regardless of their position in the list of waiters. An example illustrates this behavior: If a latch is held in UP mode, the first waiter also wants an UP mode latch, the second waiter wants an EX mode latch, and the following three want SH, EX, and SH, respectively. When the UP mode is released, not only is the first UP waiter granted, but the two SH requests are granted, too, even through they arrived after the first EX request. This does not cause starvation, because no grants are made unless the first waiter can be granted.

Latch Wait Time

The wait time displayed in `sys.dm_os_waiting_tasks`, and the averages derived from `sys.dm_os_wait_stats`, for latch waits is misleading. The wait time used in these locations is how long the task has been idle waiting for the latch. Latch waits, however, wake up every five minutes to check for signs of a problem with the latch. This check resets the wait time, so no latch wait ever shows having a wait time longer than five minutes. It is important to note that this does not mean that the logical duration for a latch wait never exceeds five minutes; it just means that this total duration is made up of units of at most five minutes. Although the full logical wait time for an individual latch wait is not available from a DMV, full logical average and maximum durations are available from the `sys.dm_os_latch_stats` DMV.

Latches were intended to be held for only short durations, and this is usually the case. However, in severely overburdened systems and in a few other rare cases, it is possible that a latch can take an extended amount of time to acquire—it cannot be acquired within five minutes. When this occurs, SQL Server writes a warning message in the error log. This warning is commonly referred to as a latch timeout warning. For nonpage latches, this is a purely informative message. Noncritical page latch waits abort with an 845 exception if the latch request is no closer to being granted than it was at the start of the five-minute duration.

Page Latches

Page latches are used to provide synchronization on physical access to individual database pages that are stored in the buffer pool. Access to on-disk database pages is controlled by the buffer pool; thus, page latches effectively provide access to the on-disk pages.

Resource Description

The resource description in the `resource_description` column of `sys.dm_os_waiting_tasks` for all page latches is `<dbid>:<file-id>:<page-in-file>`. The `<dbid>` is the database ID for the database to which the page belongs. The `<file-id>` is the ID of the file within the database. This corresponds to the `file_id` column in the `sys.database_files` catalog view. `<page-in-file>` corresponds to the `<page-in-file>`th page in the file.

PAGELATCH Versus *PAGEIOLATCH*

The difference between the two wait type subgroups for page latches is minor. For a given page ID, waits for a latch may use a wait type for either subgroup. The choice is determined by whether an IO operation is outstanding on the page at the time the latch wait starts. The wait time is updated only every five minutes during a latch wait, corresponding to the timeout check described earlier.

Latch Mode Usage

To read a database page, at least an `SH` latch is required. Writes to a page generally require an `EX` latch; exceptions to this rule are internal allocation system pages and versioning pages in tempdb that require only `UP` mode latches.

Causes of Blocking on Page Latches

There are four main causes of page latch blocking:

- IO subsystem performance
- Contention on internal allocation system tables
- Contention on catalog pages
- Contention on data pages

The first cause manifests itself as `PAGEIOLATCH` waits. This is an indication that the IO subsystem cannot keep up with the IO load. This may be caused by a malfunctioning IO subsystem or excessive, and possibly unnecessary, load on the IO subsystem. When observing the `PAGEIOLATCH` waits, the first thing to check is the duration of the waits. Very long waits are signs of a malfunctioning IO subsystem; short durations are more likely to be a sign of high IO load. In the middle are the troublesome ones that do not clearly belong in either camp. SQL Server defines a long IO as one taking more than 15 seconds. If such an IO is encountered, a message is written to the error log. Note that this error message occurs only once every five minutes for a given file. The message includes a count of the lengthy IOs seen for file in question during the past five minutes:

```
SQL Server has encountered 1 occurrence(s) of I/O requests taking longer than 15
seconds to complete on file [D:\SQL\DATA\tempdb.mdf] in database [tempdb] (2). The
OS file handle is 0x00000638. The offset of the latest long I/O is:
0x000000ffd00000
```

You can use the sys.dm_io_pending_io_requests DMV to determine where a pending IO is held up. The io_pending column indicates whether the operating system has marked the IO as completed. For long IOs, the next step is to examine where the IO is stuck. If the io_pending value is 1, the operating system has not completed the IO. These cases require investigating the operating system and/or IO subsystem for the cause of the delayed IO. Note that the duration of PAGEIOLATCH waits should not be used as a direct measure of IO duration; this also applies to wait statistics. This is because these latch waits start when someone attempts to wait for the completion of the IO, *not* when the IO was originally issued. Similarly, at the completion of the IO, the latch wait does not end until the latch has been granted. This could be some time after the completion of the physical IO, because there could have been multiple requests for the latch, and later requests would have to wait for earlier requests to be granted first. For similar reasons, the wait count counters for these waits do not indicate the number of IOs that were waited on, because multiple waiters can wait for the same IO to complete.

Many short IO waits are most likely caused by an overloaded IO subsystem. This requires investigation of what is causing the IO overload. One possible cause is lack of memory relative to the size of the application's working set of database pages. Not all pages accessed by an application need to be in memory, but if a significant fraction of the most often used pages do not fit in the buffer pool, they are likely to be constantly read from disk. A related cause is excessive and unnecessary IO activity from SQL Server, such as that caused by table scans on large tables when the query could benefit from an index to avoid the scan. These scans can cause unnecessary thrashing in the buffer pool, which is characteristic of the insufficient memory case. Therefore, it is important to rule out unnecessary buffer pool thrashing due to less-than-ideal access patterns before determining whether more memory is required. The sys.dm_exec_query_stats and sys.dm_db_index_operational_stats DMVs can be of help with this. The former contains counters for both logical and physical reads and for logical writes.[2] The latter contains counters for PAGEIOLATCH waits and wait times on a per-index basis. These stats can point toward queries and plans that perform a lot of IO and therefore might be worth investigation as to whether all that IO really needs to be generated. Similarly, indexes with heavy IO loads can be identified:

```
select sql_handle, plan_handle,
       total_physical_reads, last_physical_reads,
       min_physical_reads, max_physical_reads,
       total_logical_writes, last_logical_writes,
       min_logical_writes, max_logical_writes,
       total_logical_reads, last_logical_reads,
       min_logical_reads, max_logical_reads
from sys.dm_exec_query_stats
select database_id, object_id,
index_id, partition_number,
       page_io_latch_wait_count,
```

2 Queries never directly cause physical IO in SQL Server; therefore, physical IO counters are not included. All physical data page IOs are issued by the buffer pool as part of managing the set of cached pages.

```
page_io_latch_wait_in_ms
from sys.dm_db_index_operational_stats (null, null,
 null, null)
```

The remaining causes do not require IO activity; rather, they are caused by high concurrent activity on specific pages. Differentiating among the three requires determining which category the page ID falls under. The internal allocation system pages are the easiest to determine; the others need a bit more work.

The internal allocation system pages are at fixed intervals in each file, so they can be identified using simple calculations on the page-in-file portion of the page ID. The set of pages are the PFS, GAM, and SGAM pages. The first PFS page is always page-in-file ID 1; after that, it is always a multiple of 8088. The GAM and SGAM pages are page-in-file IDs 2 and 3, and thereafter occur every 511,232 pages, which is approximately 4GB worth of disk space per file.

Another alternative for identifying these pages is the latch mode requests. As mentioned earlier, for page latches, the `UP` mode is used almost exclusively for these internal allocation system pages. Thus, `PAGELATCH_UP` and `PAGEIOLATCH_UP` waits can be assumed to be for these internal system pages. These pages are used for tracking the allocation status of pages with each file. Contention on them reflects lack of file parallelism in a file group. Thus, contention on these pages can be reduced by adding more files to the file group. This is especially the case in tempdb. You can read more about tempdb-specific troubleshooting in Chapter 9, "Tempdb Issues."

As mentioned previously, distinguishing between the remaining two causes requires knowing the object to which a page belongs. Although there is no documented way to determine this link, you can use the undocumented `DBCC PAGE` command for this purpose. As with all undocumented commands, Microsoft does not provide support for usage of this command. Use it at your own risk. That said, `DBCC PAGE` is widely used. As its name implies, it provides information regarding a database page. Its can write its output in either text or rowset format. The former is more convenient to read, whereas the latter is easier to process programmatically. For the purposes of determining the object associated with a page, you can use the following snippet. It returns four rows, one each for the object ID, index and partition number, and allocation unit ID. An allocation unit is a set of related pages tracked as a unit. It corresponds to the set of pages tracked by a single IAM chain.

```
declare @dbccpage table (
      ParentObject sysname,
      Object sysname,
      Field sysname,
      VALUE sysname)
insert into @dbccpage
      exec ('dbcc page (<dbid>, <file-id>,
                  <page-in-file>) with tableresults')
select Field, VALUE
from @dbccpage
where Field like 'Metadata:%'
```

Although contention on catalog table pages is not common in most databases, it does sometimes occur and is a sign of a high volume of DDL operations, because these operations modify catalog tables. This is most common in tempdb, where heavy usage of short-lived temp tables or table variables results in heavy catalog table activity. Contrary to popular belief, table variables are not fully in memory but have the same storage semantics as temp tables. As with contention on internal system allocation pages, resolution for catalog table page contention is discussed in Chapter 9. The key concept is to reduce the number of DDL operations by changing temp table and table-variable usage patterns so that either the tables are cachable or the usages are removed or reduced.

This leaves just user table latch contention. This contention occurs because of simultaneous access attempts on index or data pages in a table. The contention is generally on one or more hot pages. Hot pages are pages that are frequently accessed within conflicting modes, such as a combination of read and write access or multiple concurrent write accesses. Although multiple concurrent read accesess also make a page hot, they do not cause blocking, because read accesses require only `SH` latches, which are compatible with each other. This means that select queries by themselves do not cause page latch blocking, so the contention is caused by data modifications—by insert, update, and delete operations on the pages.

The resolution for this contention requires examining the reasons for the simultaneous access. This requires examining the table and index schema and data modification patterns. One common cause is concurrent insert activity into an index where inserts are adjacent, or nearly adjacent. Examples of these are identity columns as the first key of an index, or a datetime column that is populated by the current time. Because the rows are adjacent in the index, they are likely to be placed on the same page. Concurrent inserts to the same page are serialized by the required `EX` latch on the page. This might result in significant contention on the page latch if there are many concurrent inserts. This contention can be further exacerbated by another factor: page splits.

Page splits occur when a new row is inserted onto a page that is already full and the content of the existing page is split into two pages. During splits, latches are held on the page that is being split, the existing next page in the index, and the parent page in the index tree for the duration of the split. The latches are held in `EX` mode, thus blocking even read access to the pages. A split needs to allocate a new page and thus is not necessarily an instant operation. Attempts to access any of these pages during the split become blocked.

One solution to the insert point contention problem is to reorder the index keys so that the insert activity is distributed across multiple regions in the index. For an index of any significant size, this also distributes the activity across multiple pages. Of course, any changes to the order of index keys may result in application performance degradation, because the index might no longer be useful for the types of queries used in the application. If the contention is on a clustered index, in certain circumstances another option is to replace the clustered index with a heap (that is, to drop the clustered index). This might help as SQL Server automatically distributes heap inserts across multiple pages. Removal of the clustered index might also hurt application performance and should be done only after you have considered the trade-offs between clustered indexes and heaps.

Choosing Which Page Latch Contention to Focus On

It is unlikely that a few isolated page latch waits for a particular table will warrant extensive examination of the table and index schemas or access patterns—it would probably not be cost-effective, and some contention is bound to happen in any busy system. A better approach is to gather statistics based on the table/index affected by the page latch waits. Although this can be accomplished using `sys.dm_os_waiting_tasks` and `DBCC PAGE`, it is not a trivial task. Part of the reason for this is that, for programmatic processing, the `DBCC PAGE` output needs to be stored in a temporary table or table variable. More important, the page ID in object and index ID lookups needs to be performed at the time of the `sys.dm_os_waiting_tasks` query, because the pages could become reallocated to some other object if the lookup is delayed. Fortunately, the `sys.dm_db_index_operational_stats` DMV insulates you from many of the details. It contains several columns of interest, particularly `page_latch_wait_count` and `page_latch_wait_in_ms`. You can use deltas of these statistics to find indexes that are experiencing significant latch wait times or counts and focus the investigation on those indexes.

Nonpage Latches

In addition to being used for physical access control on database pages, latches are used for a variety of other synchronization needs within SQL Server. In fact, there are more than 100 unique nonpage latch usages. It is not possible to discuss all of these here, nor would it provide much useful information, because some are used extremely rarely. Therefore, I discuss some of the more common ones and provide general guidance for the rest. Although `TRANMARK` latches have unique wait type status within this group, that is only for backward compatibility, and they can be treated as any other nonpage latch.

Latch Classes

The different types of nonpage latches can be distinguished by their latch class. The latch class is an identifier attached to each latch that indicates is usage scenario. It can be used to group latches used for different purposes. The full list of latch classes is available from `sys.dm_os_latch_stats`. This DMV also provides statistical information similar to `sys.dm_os_wait_stats` for each of the latch classes. It is worth noting that latch classes apply to page latches, too; they are the `BUFFER` latch class. Because of their distinct and important usage, however, page latches are treated uniquely. The names of the latch classes were designed to provide some ability to infer the purpose of the latch or when it may be acquired. The names are generally in two or three parts: component, optional subcomponent, and usage within the component.

Resource Description

The resource description for nonpage latches provides arguably less information than page latches. The format for `TRANMARK` latches is a GUID. The GUID is the transaction UOW (unit of work) identifier for the owning task's current transaction at the time of the latch acquire. Latches are transactional, so this GUID just provides guidance as to which task acquired and holds the latch. This can be mapped to the `transaction_uow` column in

the sys.dm_tran_active_transactions DMV. For all other nonpage latches, the convention is <latch-class-name> (<memory-address-for-the-latch>). This allows grouping latch waits by latch class as well as identifier and by which waits are on different instances of latches of the same class using the memory address. For example, each in-memory data structure that represents a database file contains a latch, of class FCB, so the address allows for determining whether the FCB latch contention is all on a single file or multiple files. The next logical step is to map this FCB latch to a database and file ID. Unfortunately, there is no way to do that mapping currently, but hopefully it will be considered in a future release of SQL Server.

Latch Class Descriptions

The next sections contain short descriptions of a handful of the latch classes on which latch contention is commonly seen. It is not a comprehensive list. For those that are not listed, the following generic latch investigation steps can be used:

1. Check the latch class description in Books Online, if available.
2. Examine the latch class for indications of what components or what type of statements may acquire the latch.
3. Examine the blocked task's current statement for more hints regarding the latch usage.
4. Use the suffix of the wait type to determine the mode for the blocked request. This will help identify whether the latch is being acquired for shared access—in which case, it must currently be held for exclusive access. An exclusive request implies that the blocked task's current session needs to do some update, and this can provide further info regarding resolving the blocking.

FCB, *FCB_REPLICA*, *FGCB_ADD_REMOVE*, *FILE_MANAGER*, *FILEGROUP_MANAGER*

These latch classes are all related to various aspects of database file management. FCB stands for File Control Block. The FCB latch is used to synchronize standard database files, whereas the FCB_REPLICA latch class is used by objects that represent the sparse files used for database snapshots.

The FGCB_ADD_REMOVE class is used to synchronize files within a file group. (FGCB stands for File Group Control Block.) It is acquired in SH mode during operations such as selecting a file from which to allocate. Operations such as adding or removing a file from the file group need to acquire it in EX mode. File grow operations also need to acquire the latch in EX mode. A file grow operation thus blocks not only other file grow operations but possibly other allocations, too. Because the latch is held for the duration of the operation, large file grow operations can cause a lot of blocking. For data files, SQL Server 2005 can make use of the Windows ability to instantly initialize a file to a given size, thus improving file growth performance dramatically. These performance optimizations cannot, however, be used for log files or on older versions of Windows. In these cases, the duration of the file grow operation is dependent on the size of the file grow and the amount of other IO activity.

ALLOC_EXTENT_CACHE **and** *ALLOC_FREESPACE_CACHE*
These classes are used to synchronize access to caches of extents and pages with available spaces. The extent cache is used by all HoBts, whereas the freespace cache is used by only heaps and lobs. Contention on these caches can occur during extremely high concurrent insert, update, and delete operations on an index or heap where the operations require new space within a page or a new page to be allocated. This contention is not affected by IO performance. If contention is significant, a possible solution is to partition the table so that the insert/update/delete operations are spread across the partitions. This helps because these caches are specific to a single HoBt, and partitions are implemented using multiple HoBts.

*APPEND_ONLY_STORAGE_** **and** *VERSIONING_**
These groups of latch classes are related to row versioning. The `APPEND_ONLY_STORAGE` group of latch classes is used for synchronization by append-only storage units. These are special allocation structures that exist only in tempdb and are heavily used by row versioning to store previous versions of rows. The `VERSIONING` group is used for state transitions and transaction management.

*ACCESS_METHODS_**
This group of latches is used by the Access Methods component of SQL Server. This component handles the access paths to reach data; for example, it navigates the index and heap structures to reach the appropriate rows based on the query predicates.

Individual classes worth mentioning within this group are `HOBT_COUNT`, which is used to synchronize updates to row and page counters, and `DATASET_PARENT`, `KEY_RANGE_GENERATOR`, and `SCAN_RANGE_GENERATOR`, which are specific to parallel plans. Cache-only HoBts represent HoBts that do not appear in the system catalog, such as work tables, and thus are not persisted across SQL Server restarts.

*TRACE_**
Classes within the trace group are used during SQL Server tracing, such as through Profiler. Contention caused by waits on these latches can be reduced by reducing or disabling trace activity.

LOG_MANAGER
With this class, it is important to note that it is not used for basic transaction log operations and thus does not affect mainline log throughput. It is, however, used to synchronize log file grow operations. Thus, an option for resolving contention on this latch class is to size the log file appropriately upfront or monitor log file usage and manually grow the file during slow periods.

*TRANSACTION_**, *MSQL_TRANSACTION_MANAGER*, *NESTING_TRANSACTION_FULL*, *NESTING_TRANSACTION_READONLY*
Within this group of latch classes that are used during various transaction-related operations, the `TRANSACTION_DISTRIBUTED_MARK` latch is unique. It is used when placing markers in the transaction logs to allow for recovery to a named point. There is only one

transaction mark latch in any instance of SQL Server 2005. This latch rarely, if ever, encounters contention, and thus there is no need for an extensive description. The source of any contention is also clear, because this latch is used by only a single operation. The other latch classes in this group are used in various transaction contents.

Locks

Blocking on lock resources is perhaps one of the most common causes of blocking. Although much information is available for investigating lock blocking, the process is made tricky by the fact that locks are one of the resources that can be held across batches. This is an important aspect to remember when investigating lock blocking and one that I cannot emphasize enough: *Locks held by a transaction may not have been acquired by the current statement within the transaction*. Waits on locks use one of the `LCK_M_*` wait types. The suffix of the wait type name is the lock mode requested by the blocked process.

Lock Resource Definition

Locks are acquired on lock resources. A lock resource is just a set of values that identifies the resource. Although these resources are divided into 11 groups based on the logical object being locked, no physical connection exists between the lock and the object. Therefore, it is possible to acquire a lock on an object that does not exist. This is important to keep in mind when attempting to query for additional information on a certain lock resource. The list of groups is available in Books Online under the `sys.dm_tran_locks` topic. In addition to the lock resource type, lock resources contain database ID and resource-type-dependent information. The size of the type-dependent data is limited, so some resource types cannot store full uniquely identifying information for a particular resource. This means that it is possible to have false collisions between locks for different resources of the same type. It is, however, impossible to have false collisions for locks in different databases or of different resource types, because that information is uniquely available for all lock resources. Some lock resource types also have subtypes. It is important to note that the type-subtype pairs do not form a hierarchy and specifically do not use multigranular locking. Subtypes only further scope the lock resource and are used for resources that are related to the main type.

A lock resource is described in readable form in the resource description of the `sys.dm_os_waiting_tasks` DMV and `sys.dm_tran_locks`. Note that the formats of the resource descriptions in these DMVs are new to SQL Server 2005. These new formats are described for each resource type in the section for that resource type.

Lock Grant/Blocking Order

It is often useful to know the order in which locks are granted, because this can help you understand why certain blocking occurs. This order also affects lock blocking chains, because it determines which tasks appear first in the blocking chain. Lock requests can be in one of three different states: granted, converting, or waiting. Requests in the `GRANT` state have been granted. (That is, the lock is held in that mode.) Requests in the `CONVERT` and `WAIT` states have not yet been granted. In both cases, the requestor is waiting for the

lock to be granted. The difference between the two states is that in CONVERT, the requestor already holds a lock with a weaker mode on the resource, whereas in WAIT, it owns no lock on the resource. Conversion happens, for example, if a transaction first reads a row, resulting in an S lock, and then proceeds to update the row, which requires an X lock. Note that the update must happen before the S lock is released for conversion to occur. A conversion attempt simply converts (or upgrades) the existing grant mode; it does not result in a new entry in sys.dm_tran_locks when the conversion has been granted. In the following example, the first sp_getapplock acquires an S application lock on the resource amalgam-demo-lock, and the second converts it to an X:

```
begin tran
exec sp_getapplock 'amalgam-demo-lock', 'Shared'
select *
from sys.dm_tran_locks
where request_session_id = @@spid
exec sp_getapplock 'amalgam-demo-lock', 'Exclusive'
select *
from sys.dm_tran_locks
where request_session_id = @@spid
rollback
```

The order of blocking chains on a single lock resource is as follows:

- If a waiter is not the first waiter with state WAIT, it is always considered blocked by the first waiter with state WAIT. This because it cannot be granted before the first waiter is granted.
- If a waiter is the first waiter with state WAIT and at least one waiter has state CONVERT, the waiter is considered blocked by the conversion requests. This is because conversion requests have priority over nonconversion requests, so the waiter cannot be granted until the converter has been granted.
- If a waiter is the first waiter with state WAIT and no waiters have state CONVERT, it is considered blocked by all granted requests with incompatible lock modes.
- A waiter with state CONVERT is considered blocked by all granted requests with incompatible lock modes.

Note that if a waiter is considered blocked by multiple other tasks or sessions, sys.dm_os_waiting_tasks displays each of the blockers in a separate row. This is different from what sys.dm_exec_requests or the deprecated sysprocesses show; they show only the first blocker.

Locks are granted in a relaxed first-in, first-out (FIFO) fashion. Although the order is not strict FIFO, it preserves desirable properties such as avoiding starvation and works to reduce unnecessary deadlocks and blocking. New lock requests where the requestor does not yet own a lock on the resource become blocked if the requested mode is incompatible with the union of granted requests and the modes of pending requests. A conversion

request becomes blocked only if the requested mode is incompatible with the union of all granted modes, excluding the mode in which the conversion request itself was originally granted. A couple exceptions apply to these rules; these exceptions involve internal transactions that are marked as compatible with some other transaction. Requests by transactions that are compatible with another transaction exclude the modes held by the transactions with which they are compatible from the unions just described. The exclusion for compatible transactions means that it is possible to see what look like conflicting locks on the same resource (for example, two `X` locks held by different transactions).

The FIFO grant algorithm was significantly relaxed in SQL Server 2005 compared to SQL Server 2000. This relaxation affected requests that are compatible with all held modes and all pending modes. In these cases, the new lock could be granted immediately by passing any pending requests. Because it is compatible with all pending requests, the newly requested mode would not result in starvation. In SQL Server 2000, the new request would not be granted, because, under its stricter FIFO implementation, new requests could not be granted until all previously made requests had been granted. In the following example, connections 1 and 3 would be granted when run against SQL Server 2005 in the specified order. In SQL Server 2000, only connection 1 would be granted:

```
/* Conn 1 */
begin tran
exec sp_getapplock ‘amalgam-demo', 'IntentExclusive'
/* Conn 2 */
begin tran
exec sp_getapplock 'amalgam-demo', 'Shared'
/* Conn 3 */
begin tran
exec sp_getapplock 'amalgam-demo', 'IntentShared'
```

When Was a Lock Acquired?

As mentioned previously, locks may have been acquired by statements prior to the current statement, so the question of determining when a lock was acquired is a common one. Unfortunately, there is no guaranteed way to determine which statement acquired a lock, or even when a lock was acquired. It is sometimes possible to rule out the current statement as the acquirer. You can do this by comparing the wait time for the task that is blocked on the lock and the start time of the owner's last batch. If the lock owner's current batch has been running for a shorter time than the waiter has been waiting, the lock could not have been acquired by the owner's current statement. Here's an example:

```
select *,
case
when getdate () >
DateAdd (ms, wt.wait_duration_ms,
                          es.last_request_start_time)
           then 'yes'
           else 'unknown'
     end as blockers_past_statement_acquired_resource from
```

```
      amalgam.dm_os_waiting_tasks_filtered wt left join
      sys.dm_exec_sessions es
      on es.session_id = blocking_session_id
```

Although this can be used to rule out the current statement, it does not help identify the specific previous statement that acquired the lock. The only way to reliably determine which statement acquired a lock is to examine all the statements executed within the owner's current transaction and analyze them to determine which one(s) would have needed to access the locked resource. This process is complicated by the fact that a transaction may have started earlier than the application developer planned. This can occur, for example, if a previous transaction was not terminated correctly. A simple case of this occurs when the application contains a bug that causes it to erroneously neglect to terminate a transaction. However, a more subtle cause occurs when a statement or batch is aborted and the application assumes the transaction has also been aborted, but, in fact, the error wasn't severe enough to abort the transaction. This is commonly caused by the client sending an abort request to the server. The transaction-related DMVs can be used to help detect cases such as these where a transaction has been active longer than expected. The following query, for example, shows the open transaction count and transaction name and start times for every session with an active transaction. Both the open transaction count and the transaction name can be of use in determining that a transaction has not been terminated properly. The open transaction count is incremented with each begin transaction, so a value greater than expected for the application's current location is a hint of a possible problem. Even clearer is a transaction name mismatch. This is, in fact, a good reason to use named transactions in applications. Here's a sample query:

```
select er.session_id, er.request_id,
      er.open_transaction_count, er.transaction_id,
      at.name, at.transaction_begin_time
from sys.dm_exec_requests er join
      sys.dm_tran_active_transactions at
      on er.transaction_id = at.transaction_id
```

When attempting to match a lock with a statement in a transaction, it is useful to know what lock mode the lock is held in by the owner. A simple way to do this is to look at the mode attribute in the `sys.dm_os_waiting_tasks.resource_description`. This mode corresponds to the sum of all granted lock modes on the resource. When there are multiple blockers, this value is the same for each of them, even if they acquired different lock modes.

Another technique is to get each owner's individual granted lock mode from `sys.dm_tran_locks`. To do this, the resource description attributes need to be mapped to lock resource identification information in `sys.dm_tran_locks`. The resource identification columns are all the columns with the `resource_ prefix`. As a unit, these columns uniquely describe a lock resource:

```
select request_mode
from sys.dm_tran_locks
```

```
where request_session_id = <blocker-session-id> and
request_execution_context =
<blocker-execution-context> and
resource_database_id =
<resource-dbid> and
resource_type =
<resource-type> and
resource_subtype =
<resource-subtype> and
resource_lock_partition =
<resource-lock-partition> and
resource_associated_entity_id =
<resource-associated-entity-id> and
resource_description =
<resource-other-desc-as-in-tran-locks>
```

Armed with the blocker's lock mode, it is easier to identify the statement that might have acquired the lock. Share-type lock modes may be acquired by any statement. Even DML statements may acquire such locks as part of subqueries or to qualify rows. Exclusive-type lock modes are usually acquired only by DML queries, although regular queries can also acquire them when a locking hint such as `XLOCK` is specified. `SCH_M` locks are acquired only for DDL operations that include table truncations. Even queries running under the read uncommitted isolation level or with `NOLOCK` hints acquire some locks. Specifically, they can acquire metadata locks while compiling the query and `SCH_S` locks on the objects used. The `SCH_S` locks are required even under these isolation-level requirements to block DDL operations for the duration of the query. Otherwise, the object could be dropped while the query is executing. Key range locks are acquired only by statements executed under the `SERIALIZABLE ISOLATION` level. This can help identify the statements responsible for acquiring the locks, because most applications do not make heavy use of this isolation level. Note that the `HOLDLOCK` locking hint is equivalent to the `SERIALIZABLE ISOLATION` level.

Although most locks are transaction scoped and are released when the transaction terminates, it is possible to hold locks across transaction boundaries. In these cases, the locks are owned by some other entity. The owning entity is available from the `sys.dm_tran_locks.request_owner_type`. Cursors and sessions are examples of such entities. For locks held by sessions, the scope of investigation expands to the start of the owner's session. Fortunately, only a limited number of lock types can be acquired at the session level; of those, only two are ever held across transaction boundaries—database and application locks. It is easy to identify which statements would have acquired these locks. Session-level database locks are acquired only by `USE` statements. Session-level application locks can only be acquired using the `sp_getapplock` stored procedure. Remember, however, that the `sp_getapplock` call could be made from another stored procedure. The statements responsible for acquiring the locks held by a cursor are also relatively easy to determine, because they could only have been acquired during cursor operations (for example, `FETCH`).

Although there is no way to accurately determine, in every case, which statement acquired a particular lock after the fact, it is possible in test environments with a little forethought. You can do this by capturing the `Lock Acquire` trace events along with `Statement Start` and `End` trace events. Note that this can produce an extremely large quantity of trace output and will likely cause performance degradation for the statement and the server, so the `Lock` trace event should not be used in production systems. Also note that the specific locks acquired depends on several factors, including the query plan and other concurrent operations. Query plans affect the locks acquired, because different plans may examine more or fewer lockable resources while producing the result set. Concurrent access can also result in the acquisition of different or varying numbers of locks. One example of a runtime behavior difference is an optimization used during read committed isolation-level queries that safely skips the acquisition of row locks if there have been no modifications to the page since the start of the transaction. When tracing multiple statements within a transaction, you can also use the `Lock Release` trace event to exclude locks that are released at the end of a statement.

When using lock tracing, it is advisable to filter the trace output based on the session ID for the connection on which the statement(s) of interest will be executed. This is because even on a relatively idle server, background system tasks acquire locks and cause trace output. Similarly, the trace should be started after the connection has been established so as not to populate the trace with events for locks acquired during the login phase. Also, ideally the statement will have been compiled and cached prior to the traced execution, because this further helps limit the amount of output to examine.

Lock Resource Descriptions

The next sections cover a selection of lock resources commonly seen in blocking situations. It is not a complete list, but the `sys.dm_tran_locks` topic in Books Online has good descriptions of the remaining resources.

Object

The `objectlock` resource is used for locking database objects. These objects can be tables, stored procedures, views, triggers, or any other object with an object ID as listed in the `sys.all_objects` catalog table. Object locks appear in blocking mainly in two flavors—object-level locks on tables that cause blocking and blocking on the `COMPILE` subresource.

The first step is to determine the identity of the object on which blocking is occurring. The format of the resource description in `sys.dm_os_waiting_tasks` for object locks is as follows:

```
objectlock
lockPartition=<lock-partition-id>
objid=<object-id>
subresource=<sub-resource-name>
dbid=<database-id>
id=lock<lock-memory-address>
mode=<sum-of-all-granted-owners>
associatedObjectId=<associated-object-id>
```

The non-self-explanatory fields are explained next. The `<sub-resource-name>` indicates the subresource used for this lock. A value of `FULL` indicates that the full resource is used, not the subresource. The `<associated-object-id>` is in fact the object ID, and it corresponds to the `resource_associate_entity_id` column in `sys.dm_tran_locks`.

The name of the object can be retrieved using the `OBJECT_NAME` function. Note that this function operates on the current database, so it must be used from the appropriate database for it to return accurate results, because the same object IDs likely refer to different objects in different databases:

```
SELECT OBJECT_NAME (<object-id>)
```

Alternatively, you can use the `sys.all_objects` catalog. This has the benefit of displaying the type of object too:

```
SELECT name, object_id, schema_id, type_desc
FROM sys.all_objects
WHERE object_id = <object-id>
```

Because of the use of multigranular locking in SQL Server, most statements need only an intent lock at the table level. (Refer to the Books Online topic "Lock Modes" for more information on intent and nonintent locks.) Because all intent locks are compatible with each other, under most circumstances blocking does not occur at the table level. Blocking occurs only if a statement requires a nonintent lock and conflicting locks are held by other transactions, or if the table is already held in a nonintent mode when an intent mode request is made. In both cases, avoiding the nonintent mode lock is the best option for resolving the blocking.

Three nonintent modes are responsible for most table lock blocking. The first are the `SCH_M` lock mode, the strongest possible mode, which is used for schema modifications. `S` and `X` locks may be caused by lock escalation, locking hints, the query plan or relevant statistics operations, or index options.

Index options can result in table-level locks if both page and row locking are disabled on an index. You can use the `INDEXPROPERTY` function to determine whether this is the case. Note that because an object lock covers an entire table, including all indexes, a single index with both page and row locking disabled will affect access to the entire table if the index is accessed.

A locking strategy is calculated for a query when it starts. This calculation determines what locking granularity the query should use. This calculation may determine that even with the decreased concurrency, it would be beneficial for the query to acquire a single table lock rather than thousands or millions of page and row locks. Note that these calculations are biased against table-level granularity because of potential negative concurrency effects. In certain cases, however, a table lock is the best option. Fortunately, a table-level granularity choice is only a best-effect choice—if the nonintent lock cannot be acquired instantly, the query backs off to page- or row-level locking. However, if there is no

conflict at the start of the query, the table-level lock is granted. Often, these table-level granularity choices are caused by nonideal or out-of-date table statistics that skew the calculations in the table granularity direction. That said, it is of course certainly possible that the query plan is valid and a table lock is a good choice, in which case locking hints can be used to avoid the table-level lock if the query results in blocking. Either a page or row locking hint can be used.

A related cause of table-level locks are heaps (tables without clustered indexes) and the `SERIALIZABLE ISOLATION` level. Under the `SERIALIZABLE ISOLATION` level, any scan of a heap requires a table-level lock for maintaining isolation-level characteristics. These locks can be avoided either by not using the `SERIALIZABLE ISOLATION` level or by querying a heap through a secondary index rather than directly. Such a query acquires the appropriate locks on the index, and the access to the heap goes directly to the appropriate page and row, thus not requiring a table-level lock.

On the flip side, locking hints can also cause table-level locks. This is the case if the `TABLOCK` or `TABLOCKX` hints are used. These instruct query execution to acquire either an `S` or an `X` lock on the table, respectively. As opposed to table locks that are suggested by granularity calculations, table lock requests based on locking hints wait until the lock can be granted and thus are much more likely to cause blocking.

The final factor that can result in table-level locking is lock escalation. Lock escalation is the process of a statement escalating from using page or row-level locking granularity to table lock granularity based on a system observation that the statement is acquiring a large number of locks on the table. The trigger point for lock escalation is when a single query has acquired at least 5,000 locks on a single partition. Note that for self joins, each "instance" of a partition is counted separately. When lock escalation occurs, previously acquired page and row locks for the table in question are released after a table-level lock has been acquired. The intent of this is twofold: reduce the memory requirements of large queries (each lock takes about 96 bytes of memory) and slightly increase performance by avoiding the execution of the locking code completely. The attempt to escalate is best-effort; that is, if the table-level lock cannot be acquired, lock escalation is skipped. In certain types of workloads, it is fairly common for lock escalation to occur. Although the only way to determine whether lock escalation has occurred in a particular query is to run a trace that includes the `Lock: Escalation` event, it is possible to determine statistically which tables are likely to experience lock escalation based on their query patterns. This is done using the `index_lock_promotion_attempt_count` and `index_lock_promotion_count` columns in `sys.dm_db_index_operational_stats`. Lock promotion is a synonym for lock escalation. These counts tell you how many times lock escalation has been attempted and how many times it has succeeded. It is important to note that the attempt count is based on the number of times the lock manager suggests that escalation might be needed. This occurs even before the 5,000-lock-per-partition threshold is reached, so the attempt count should not be viewed as indicating that queries on this index have acquired 5,000 locks on the index but failed to acquire the table lock. A more accurate description is that the attempts count reflects the number of times the index has been used in a query where the transaction has acquired increments of 1,250 locks—the count is incremented every time a multiple of 1,250 locks is reached.

Even though it often has a bad reputation, lock escalation is not always a bad thing. The server supports lock escalation for a very good reason—to minimize the resources used to manage concurrency. However, when lock escalation affects concurrency negatively, it can be avoided either by modifying queries so that they do not acquire so many locks or by disabling lock escalation. Lock escalation can be disabled instance-wide by enabling trace flag 1211 or 1224. The difference between the two is that 1211 disables lock escalation across the board, whereas 1224 disables it only until the lock manager is under memory pressure. Lock escalation can also be prevented on a table-by-table basis with a little extra work. This involves making sure that an intent-exclusive lock is always held on the table. Because both `S` and `X` locks are incompatible with an `IX` lock, lock escalation would always fail. One way to guarantee this is to always start a process at SQL Server startup that connects to the server and within a transaction acquires a lock on the table using an update or delete statement that does not affect any rows. This connection would then need to remain idle. The connection cannot terminate, because that would release the lock. This approach is not always convenient, but it has been used successfully and is useful when lock escalation is desired on some tables but is causing blocking on others. A query like this would work, for example:

```
BEGIN TRAN
DELETE FROM <table-name> WHERE 1=0
```

Object Compile Locks

The `COMPILE` subresource of an object lock is used to synchronize compiles. Blocking on this resource indicates that multiple tasks are concurrently attempting to compile the same object. Usually, it is a stored procedure. After the object in question has been identified, the next step is to determine why the object is being compiled concurrently by multiple tasks. When the number of such compiles or recompiles has been reduced, the contention on the compile lock is alleviated.

Page, Key, and Row Locks

I have grouped these three resources because they share common traits and are closely related. As their names imply, these resources are used to lock database pages, index keys, and heap rows. Note that key locks are never acquired on heaps, and row locks are never acquired on indexes.

The resource description formats for these resources are as follows:

```
Page: pagelock
fileid=<file-id>
pageid=<page-in-file>
dbid=<database-id>
id=lock<memory-address-for-lock>
mode=<sum-of-all-granted-owners>
associatedObjectId=<associated-entity-id>
Row: ridlock
```

```
fileid=<file-id>
pageid=<page-in-file>
dbid=<database-id>
id=lock<memory-address-for-lock>
mode=<sum-of-all-granted-owners>
associatedObjectId=<associated-entity-id>
Key: keylock
hobtid=<hobt-id>
dbid=<database-id>
id=lock<memory-address-for-lock>
mode=<sum-of-all-granted-owners>
associatedObjectId=<associated-entity-id>
```

The `dbid` tag indicates the database for the resource. The `id` tag contains the in-memory address of the lock resource data structure; this is the structure shared among locks on a given resource. The `resource_address` column for locks contains the address of the lock owner structure, which is a per-lock owner structure. The `mode` tag contains the combined mode of all granted locks for this resource. For page and row locks, the `file` and `pageid` tags provide the page identity for the resource. For these two resources, the `associatedObjectId` tag contains the `hobtid`.

For key locks, the `hobtid` and `associatedObjectId` tags contain the same value—the HoBt ID of the HoBt in which the key exists. This can be mapped to a table, index, and partition using the `sys.partitions` catalog view:

```
select object_name (object_id), object_id, index_id,
     partition_number, hobt_id
from sys.partitions
where hobt_id = <hobtid>
```

Unfortunately, the "Row and Key" resource description in `sys.dm_os_waiting_tasks` leaves out two crucial pieces of information. This is regrettable considering that the DMV is otherwise an excellent and improved source of blocking information. The missing pieces are the slot ID for row locks and the key column hash for key resources. Each of these distinguishes a particular row or key resource from other resources on the same page or HoBt, respectively. Fortunately, it is relatively simple to get this information from other sources. The two options are the `sys.dm_tran_locks` and `sys.dm_exec_requests` DMVs. The `sys.dm_tran_locks` option is more reliable but is more expensive when a very large number of locks are held in the system. Conversely, the `sys.dm_exec_requests` option may be faster but is less accurate because it includes only the resource description for the main worker during parallel queries. Therefore, the key resource information would not be available if a parallel worker is blocked on the key lock.

The `sys.dm_tran_locks` option makes use of the `sys.dm_os_waiting_tasks.resource_address` column, which can be used to join with `sys.dm_tran_locks.lock_owner_address` to find the row corresponding to the pending lock request. The missing information is contained in the `sys.dm_tran_locks.resource_description` column and can be included in addition to the standard

sys.dm_os_waiting_tasks columns. This additional information is included in the amalgam.dm_waiting_tasks_filtered2 view:

```
select wt.*,
l.resource_description as
      addition_resource_description
from sys.dm_os_waiting_tasks wt left join
sys.dm_tran_locks l
      on wt.resource_address = l.lock_owner_address
```

Alternatively, the join can instead be with sys.dm_exec_requests:

```
select wt.*,
      er.wait_resource as
additional_resource_description
from amalgam.dm_os_waiting_tasks_filtered wt left join
      sys.dm_exec_requests er
      on wt.waiting_task_address = er.task_address
```

The extra information this makes available for row locks is the slot ID. The slot ID is the ordinal of the row slot on a page in which the row has been placed. Note that the extra column displays the full row ID—that is, <file-id>:<page-in-file>:<slot-id>. For key resources, the extra information is the hash of the index key values for the index row.

A common difficulty with page, row ID, and key resources is that they cannot be easily mapped to the actual row or rows that they cover. The difficulty with page resources is the page resource is a physical resource, which means the set of rows it covers can change. In fact, the page can be repurposed for another HoBt when it is no longer needed by the current HoBt. The result is that page resources have no permanent relationship to rows in a table except as the current storage for the rows. The same is the case for rows that are identified by a page ID and slot ID on the page. Page and Row ID resources can become associated with a different HoBt when a page becomes empty and is deallocated. At this point, it can be reallocated to some other object. The HoBt ID that is provided for page and row resources is extra information that is not always available. This is because it needs to be provided at the time the lock is acquired, and in some cases the code acquiring the lock knows only the page ID. As opposed to page and row resources, key resources are logical and are always specific to a particular HoBt, and the HoBt ID is an integral part of the resource identifier; a different HoBt ID changes the identity of the resource. Although key resources can always be matched to a HoBt, they cannot be easily matched to a particular index key, because the index keys are represented by a hash value in the key resource identifier. This hash is not reversible, and the system does not maintain a mapping of hash values to keys.

Although there is no built-in mapping, there are two ways to determine which rows exist on a given page and which rows correspond to a particular row or key resource. For row ID resources, there is also a third option. The first is common to pages, row, and key resources. It uses the fact that cursors can hold locks on the current page and key/row

position. Therefore, if a cursor is used to scan an entire index, it is possible to definitively match the row values with the resource identifiers of the resources currently locked by the cursor and then compare these resource identifiers with those that are being searched. Here's an example:

```
declare @key1 keytype1
declare @key2 keytype2
...
declare @page_resource_description nvarchar(512)
declare @row_resource_description nvarchar(512)
-- some page resource of the form
-- '<file-id>:<page-in-file''
select @page_resource_description = ''
-- some row resource of the form
-- '<file-if>:<page-in-file>:<slot-id>'
-- or a key resource hash of the form
-- '(<hash-value>)'
select @row_resource_description = ''
declare find_lock_cursor cursor
scroll dynamic scroll_locks
for
     select <keyname1>, <keyname2>, ...
     from <table-name>
          with (index = <index-id>)
open find_lock_cursor
fetch next from find_lock_cursor
into @key1, @key2, ...
while @@fetch_status = 0
begin
     if (exists (
          select *
          from sys.dm_tran_locks
          where request_session_id = @@spid and
                resource_database_id =
db_id (<target-database>) and
                resource_associated_entity_id =
                     <hobtid-of-interest> and
                (resource_description =
                     @page_resource_description or
                 resource_description =
                     @row_resource_description) and
                request_owner_type = 'CURSOR'))
     begin
          select @key1, @key2, ...
     end
     fetch next from find_lock_cursor
     into @key1, @key2, ...
end
close find_lock_cursor
deallocate find_lock_cursor
go
```

There are several keys to making this method work. First, the HoBt ID corresponding to the page, row, or key needs to be known. This information is generally available from the resource descriptor in `sys.dm_os_waiting_tasks`; if not, you can use one of the alternative methods. The HoBt ID then needs to be mapped to a table and an index. The table name is needed for the `FROM` clause and the latter for the index hint. This hint needs to be used because the lock resources are specific to an index partition. Although including nonkey columns in the projection does not break the algorithm, it can make the search slower, because the index covers the query, and therefore the extra columns need to be retrieved from the base table.

Although the sample code does not demonstrate this, it is possible to terminate the loop early after all matching rows have been found. The termination case for row IDs and keys is obvious: only one row can match the lock resource, so, after that row has been seen, there is no need to continue searching. For pages, the termination case relies on the fact that the index is scanned in order, which means the same page is not visited twice during the scan. Therefore, if a matching page resource has been seen and the current page resource no longer matches, all rows on the page have been seen.

Note that it is theoretically possible to have multiple distinct index keys that hash to the same value. This is rare. The size of the hash is 6 bytes, which mathematically means that only in a HoBt with more than 2^48 unique keys is such a collision guaranteed to occur. It is, however, theoretically possible to have such a false collision with just two keys. If a hash collision is suspected, the termination condition of the search can be removed to search the entire index. An output with more than two rows indicates that a collision has occurred. The effect of a collision is that attempts to lock two rows with distinct keys will conflict. Again, this false conflict is extremely rare, and other causes of blocking should be investigated before focusing on this remote possibility.

It is possible that the scan will not find any qualifying rows. This does not mean the process is broken, but rather that the row/key/page no longer exists in the HoBt. This occurs if the row has been deleted or moved or the page has been deallocated. It is also not possible to use this method while the blocking being investigated is occurring, because it relies on the ability to acquire locks on rows and pages in the index.

The second option for row and key resources is to use the `%%lockres%%` virtual column. This column contains the key hash or the row ID for index keys and heap rows, respectively. As with the cursor, an index hint is required, because `%%lockres%%` displays values for the index used. Note that this results in a full table scan unless a predicate is provided. Also, in contrast to the cursor method, this scan cannot be terminated early. The content of the virtual column matches the content of the `sys.dm_tran_locks.resource_description` column for key and row ID resources. You can use this method even when blocking is occurring by specifying the `NOLOCK` locking hint. The virtual column can produce the resource identifiers without needing to acquire the locks.

The other alternative for finding all rows on a page, and the third option for row IDs, is to use `DBCC PAGE`, which can be instructed to display the rows on the page:

```
dbcc page (1, 1, 19, 3) with tableresults
```

When using `DBCC PAGE`, it is important to verify that the page still belongs to the expected HoBt. This can be done by verifying the HoBt ID from the output by comparing the object, index, and partition IDs or names. The rows on the page are output in slot ID order. The Field column contains the column names, and the Value column contains the column's content. You can also use this method when blocking exists.

Range Locks

When using `SERIALIZABLE ISOLATION` level, it is important to understand the behavior of range locks. Range locks apply only to `KEY` locks. In SQL Server, key range locks are implemented as a special lock mode. Each range lock mode consists of two lock modes: a range mode and a specific key mode. The range mode covers the range from the key resource on which the lock is placed to the next lesser key value in the index. This range portion is exclusive of both key resources. The key portion covers the key on which the lock is placed. Thus, the combined mode is inclusive of the high key value and exclusive of the low key value. To lock the range from the highest index key to infinity, a special key resource is used. This key resource contains a hash value of `FFFFFFFFFFFF` and represents the infinity key. Any operations on key values that would exist between two existing keys coordinate with key range locking by attempting to acquire a lock on the next greater existing key. This design means that the range of key values covered by a key range lock depends on the existing keys in the index, and the number of key range locks required to lock a specific range depends on the number of existing keys within that range. Note that a key value considered existing for key range locking need not be visible to queries—keys that have been logically but not physically deleted from the index, such as ghosted records, qualify.

The key range behavior can lead to confusing situations where it appears an operation should not succeed but actually becomes blocked because of range locking. The rules for acquiring range locks are explained next.

When a predicate defines exactly one matching key, no range locks are acquired. This is because if there can be only one key with the specified values, the index must be unique, and any attempt to insert another matching key would violate the uniqueness predicate.

When a predicate may match multiple keys, range locks must be acquired for all ranges that could match the predicate. Range locks must obviously be acquired on existing keys that match the criteria. These range locks cover the range from the matching key to the next lower key. In addition, a lock must be acquired on the first key higher than the last matching key. This range lock is required to cover the range above the last existing key that currently qualifies.

The need to lock the next higher key may unexpectedly block attempts to access that range. For example, a query such as

```
select * from demo_table where a <= 4
```

would not intuitively require a lock on a key value of 10. But if only rows with a = 3 and a = 10 exist, to block inserts of keys with a = 4, the range from 3 to 10 must be locked. Therefore, take care not to miss this type of query when attempting to match locks to statements that may have acquired them.

External Wait Types

As mentioned at the beginning of this chapter, external wait types do not always indicate that the task is idle; instead, they are often used to indicate that SQL Server is executing code that may be outside its direct control. A prime example of this is an extended stored procedure call. Extended stored procedures can be written by users or third parties, and SQL Server does not have direct control over what they do. Therefore, to provide better task state information, the task is marked, for the duration of the call, as waiting for the external code to complete. This design allows for the side benefit of providing statistics on the duration of these calls—the total wait time and wait counter in `sys.dm_os_waits_stats` can be used to determine the average duration of these calls. Four wait types fall under this category:

- `MSQL_DQ`. This indicates that the task is executing a distributed query. The execution of the distributed query is outside of SQL Server's control, so the task is marked as waiting for the completion of the distributed query. Further investigation of these waits requires determining the destination of the distributed query and applying the tools available on the remote side. The destination can often be determined by examining the current statement, because it will likely be using a linked server or an ad-hoc method such as `OPENROWSET` or `OPENDATASOURCE`.
- `MSQL_XP`. This wait type occurs when a task is executing an extended stored procedure (`XP`). SQL Server does not have control over an `XP` even though it is executing within the SQL Server process. Investigation of these waits requires investigating the execution of the extended stored procedure code—the vendor may have provided diagnostic tools for the `XP`. If such tools are not available and the source code of the `XP` is not available and the documentation does not provide other troubleshooting information, contacting the vendor may be the only option.
- `MSSEARCH`. Full-text operations use this wait type to indicate that the task is processing such an operation.
- `OLEDB`. As its name implies, this wait type is used during calls to the Microsoft SQL Native Client OLEDB provider. It is also used during synchronization of certain full-text operations. Internally in SQL Server, DMVs are implemented as special OLEDB calls, and therefore any task executing a DMV-based query appears to be waiting with this wait type.

Timer and Queue Wait Types

The wait types that fall under this category do not indicate blocking. They are used by tasks for two main purposes: waiting for timers to expire before performing some periodic operation or to delay or throttle execution, and waiting for work packets for processing on a queue. Both are used almost exclusively by background system processes. Similarly, while idle with these wait types, tasks hardly ever hold any other resources on which other tasks could become blocked. Because of the nature of how these wait types are used, their associated wait times and counts can be extremely large. These high values can cause concern at first sight; however, they are perfectly normal:

- BAD_PAGE_PROCESS. This is used by a background bad page detection process to throttle its execution when running continuously for more than five seconds.
- BROKER_TRANSMITTER. When the service broker message transmitter has no messages to be processed, it waits on a queue for more work.
- CHECKPOINT_QUEUE. The database checkpoint task operates on a periodic basis. Instead of spawning a new task at every checkpoint interval, SQL Server uses a dedicated task that uses this wait type to indicate that it is waiting for the next interval, or for a new explicit checkpoint request.
- DBMIRROR_EVENTS_QUEUE. The database mirroring component uses this wait type when its work queue is empty.
- LAZYWRITER_SLEEP. This wait type is used by the background lazywriter tasks when they are suspended between work intervals. Lazywriter tasks write dirty data pages back to disk in a lazy manner; in other words, they attempt to not flood the disk subsystem with a large number of IOs.
- ONDEMAND_TASK_QUEUE. Long wait times on this wait type simply indicate that there have been no high-priority on-demand system tasks to execute. Although some background tasks, such as the deadlock monitor and checkpoint, have dedicated tasks, others share a pool of worker threads. These tasks are divided into high and low priority. The scheduler for these tasks uses this wait type when waiting for high-priority requests to arrive.
- REQUEST_FOR_DEADLOCK_SEARCH. The deadlock monitor is another background task that operates on a periodic basis and has a dedicated worker thread. In addition to period deadlock searches, other tasks can explicitly request a deadlock search. The deadlock monitor uses this wait type while waiting for the timer to expire or for explicit requests to arrive.
- WAITFOR. This is the one timer wait type that can occur while the task holds resources that could block other tasks. This is because this wait type is used for the WAITFOR T-SQL statement and thus is user-controlled. Any resources held by the connection when it executes a WAITFOR statement are held for the duration of the statement. Especially when a WAITFOR is executed within the context of a transaction, there is a risk that such resources are held.
- LOGMGR_QUEUE. The log write background thread waits on its work packet queue when it has no current work to do.
- KSOURCE_WAKEUP. When SQL Server is running as a service, a task is dedicated for responding to requests from the Service Control Manager. While waiting for such requests, the task is marked with this wait type.
- SQLTRACE_BUFFER_FLUSH. A dedicated worker is used for flushing trace buffers. This worker runs periodically, and between executions it idles with this wait type.

- BROKER_EVENTHANDLER. The Service Broker main event handler waits on its event queue using this wait type. The documentation for this wait type is somewhat misleading in Books Online, because it claims the duration should not last long; but in fact on an idle system or a system that does not use the Service Broker, this value is large by design.
- DBMIRRORING_CMD. As its name implies, database mirroring uses this wait type for a command queue.

IO Wait Types

There are several IO-related wait types. These wait types do not occur because of regular database page IO operations, because those are covered by the PAGEIOLATCH set of wait types. These other IO-related wait types apply to various other IO operations, such as log IOs, and can be either disk or network IO operations.

The LOGBUFFER and WRITELOG wait types are related to transaction logging. The latter occurs when tasks are waiting for a log flush to complete. This occurs most often during transaction commits where it is needed to maintain durability of transactions. Long wait times for this wait type generally indicate that the log disk cannot support the log volume being produced. Resolving these waits requires investigating the cause for the log disk performance problems. A common cause is having the transaction log on a shared drive. This limits the maximum performance levels, especially when the disk is used for random-access IO, such as data files. Log files are written sequentially and thus perform best when the underlying disk also can write sequentially. For highly active databases, it might be necessary to have a dedicated disk for the log. The cause of LOGBUFFER waits is similar; it occurs when no buffers are available in which to write a log record. There are a limited number of these buffers, and their unavailability indicates that the existing ones have not yet been written to the log file, allowing them to be reused.

DISKIO_SUSPEND and REQUEST_DISPENSER_PAUSE are related to external backups that freeze system IO for a moment while making a backup of the database files or drives.

Database snapshots store old copies of database pages in sparse files. Access to these pages can result in IO, which is reported with the FCB_REPLICA_READ, FCB_REPLICA_WRITE, and REPLICA_WRITES wait types. The first two indicate that multiple tasks are attempting to access the same pages in the database snapshots. The latter one occurs when a nonsnapshot statement needs to push out old copies of pages before updating the current version.

A long ASYNC_NETWORK_IO wait type is often caused by the client not processing results from the server. This causes the network buffers to fill. The server cannot send more data to the client, so the task executing the batch needs to pause while waiting for the ability to continue sending results. The fix is to change the client so that it does not leave a partially fetched result set open.

Other IO-related wait types are ASYNC_IO_COMPLETION, DBMIRROR_SEND, IMPPROV_IOWAIT, IO_COMPLETION, SOAP_READ, and SOAP_WRITE. Note that BACKUPIO and IO_AUDIT_MUTEX are not related to IO performance.

Other Wait Types

- CMEMTHREAD. This wait type occurs during synchronization of access to shared memory objects. This wait type was somewhat common in SQL Server 2000 during heavy query cache insert/delete activity because the memory for all cached query plans came from the same memory object. This has been improved in SQL Server 2005, but it can still occur. More information on memory-related investigations is available in Chapter 3, "Memory Issues."
- Parallel query wait types. Several wait types related to parallel queries are worth identifying. The CXPACKET wait type occurs when the parallel workers synchronize on a query processor exchange operator to transmit data between each other. It can indicate an imbalance in the work being performed by the tasks, and lowering the degree of parallelism may help alleviate the problem. The EXCHANGE and EXECSYNC wait types have similar causes. QUERY_EXECUTION_INDEX_SORT_EVENT_OPEN occurs during parallel index build operations.
- MISCELLANEOUS. Although common in past versions of SQL Server, this wait type should be less common in SQL Server 2005. As the name suggests, it indicates that a task is waiting for some miscellaneous reason. In SQL Server 2005, most of these unusual cases have been converted to more descriptive wait types, but several are still grouped under the MISCELLANEOUS wait type. Of these, two are worth mentioning. The first is synchronization for the NEWSEQUENTIALID built-in function. The other is synchronization of CLR assembly loads. Because these usages get clumped with each other in the MISCELLANEOUS bucket, it is not possible to differentiate between them without examining the statements being executed by the sessions.
- THREADPOOL. This wait type occurs when there are more concurrent work requests than there are workers to execute these requests. The waiting requests cannot be processed until a currently executing request completes. Depending on the expected usage levels of a system, this might indicate that the Max Workers configuration setting is too low. Whether this is the case depends on whether the currently executing requests are completing in the expected duration. Unusually long delays during the execution of requests can cause the worker pool to run out. If this is not the case, you can resolve THREADPOOL waits by increasing the Max Workers setting. However, if current statements are taking longer than average (for example, when some other blocking is causing long waits), increasing the Max Workers setting is likely to bring only temporary, if any, relief. This is because although more requests can be executed with the increased worker pool, it is likely that these requests will also execute slowly or become blocked and thus deplete the worker pool. Therefore, the key to dealing with THREADPOOL waits is to investigate and eliminate any other blocking that might be occurring.

Waits on THREADPOOL can be quite overwhelming in sys.dm_os_waiting_tasks because SQL Server SP1 considers all currently executing tasks as blocking the waiting request. This results in a large amount of output. The majority of this output can be ignored in favor of noting that a request is waiting for a worker to become available.

- SOS_SCHEDULER_YIELD. SQL Server uses cooperative scheduling. Under this scheduling model, workers are not arbitrarily interrupted by the system but are instead allowed to execute until they are forced to enter a wait state due to the unavailability of a resource or they yield voluntarily to allow other workers to execute. When a worker voluntarily yields to another worker, the yielding worker becomes idle and is effectively waiting for its turn to execute. The wait type used for this voluntary wait is the SOS_SCHEDULER_YIELD wait type, which indicates that the task yielded the scheduler and is waiting for access again. This is an expected wait type and should not be of concern; it simply indicates that the task is being a good cooperative player.

  Waits with this wait type populate the blocking task columns with the identity of the task that is currently executing on the scheduler. Until that task yields the scheduler, the blocked task will not be able to run.

A BIT OF TRIVIA

The wait type documentation in Books Online lists quite a few wait types as "Internal Only." This means that these wait types are not used in SQL Server 2005.

Deadlocks

Up to this point, I have not made much mention of deadlocks even though deadlocks are considered by many to be the ultimate in blocking. This has been intentional. In the final analysis, deadlocks are just cases of blocking that form a blocking chain with a cycle. This means that nearly everything that has been covered thus far is applicable to determining the cause and finding a resolution to deadlocks. This may sound a bit simplistic, and in certain respects it is, because deadlock avoidance may require more extensive modification than blocking avoidance. An example of this is reordering access to resources so that they are accessed in the same order so as to prevent deadlocks. However, both blocking and deadlocks can be lessened by holding resources for shorter durations, but neither is completely eliminated, because an increase in the workload could cause the blocking and deadlocking to become more prevalent.

The new deadlock output in SQL Server 2005 is far superior to the output available in previous versions. Collecting the new output does require changes to existing deadlock graph collection scripts, because it is enabled by a new trace flag. This trace flag is 1222. As with the old trace flag, the output is sent to the error log. This output can also be captured in traces, and the Profiler tool can display the deadlock graph in graphical format, which can also be saved as an XML file for more detailed analysis.

Among the improvements in the SQL Server 2005 deadlock output are the inclusion of session state, the start time of the current statement, and the transaction isolation level. Object IDs are also resolved to names when possible. The current statements of each participant are now more detailed, because they include a T-SQL call stack that shows the stored procedures and other objects in the current execution location. In addition, the SQL handle is available, so it can be used to query DMVs related to queries such as `sys.dm_exec_sql_text` and `sys.dm_exec_query_stats`.

Monitoring Blocking

The preceding sections have focused first on detecting that blocking is occurring and second on identifying the cause and possible resolutions. They have been geared more toward interactive investigations. However, it is generally not possible to dedicate a database operator to continuous active monitoring of a system. It would also not be efficient. This calls for a way to monitor blocking where alerts can be raised when blocking is encountered or the proper information gets collected automatically. To achieve this, many of the scripts from earlier sections, and some additional ones, can be rolled into a collection that can be run via SQLDiag to monitor blocking and collect the appropriate data. I have included here several script snippets and explanations as to why I would include them in a monitoring script. These can be used to build monitoring stored procedures such as `sp_blocker_pssNNN` used by SQLDiag and also available from the Microsoft website. As mentioned previously, SQLDiag is now included as part of SQL Server. This tool can collect many of the data points included here out of the box and can be extended to include custom scripts. The level of monitoring can be customized based on specific needs and the availability of CPU cycles to execute the scripts. This is an important concern because some of these script snippets can be somewhat expensive to run or might produce a lot of output that must then be analyzed.

Wait Statistics

It's always useful to collect wait statistics. These are low-impact queries. Filtering out the innocuous wait types and any zero statistics greatly reduces the output and makes it easier to review. Here's a sample query:

```
select *
from amalgam.dm_os_wait_stats_filtered
select *
from sys.dm_os_latch_stats
where waiting_requests_count <> 0
```

Current Wait Information

Various queries can be run against `sys.dm_os_waiting_tasks` to collect information on current waiters. The cheapest option is to just include the entire contents of `sys.dm_os_waiting_tasks`, or `amalgam.dm_os_waiting_tasks_filtered`, like this:

```
select * from sys.dm_os_waiting_tasks
select * from amalgam.dm_os_waiting_tasks_filtered
select * from amalgam.dm_os_waiting_tasks_filtered2
```

A slightly enhanced version includes the current statements and plans for the waiting tasks:

```
select
      amalgam.current_statement (
            st.dbid, st.objectid, st.encrypted,
st.text,
            er.statement_start_offset,
            er.statement_end_offset)
            as current_statement,
            qp.query_plan,
      wt.*
from amalgam.dm_os_waiting_tasks_filtered wt
      left join sys.dm_exec_requests er
            on wt.waiting_task_address = er.task_address
      outer apply
sys.dm_exec_sql_text (er.sql_handle) st
            outer apply
sys.dm_exec_query_plan (er.plan_handle) qp
```

And a further enhancement includes the blocking task's current statement and plan. Remember: Locks might have been acquired by a statement other than the current statement:

```
select
      amalgam.current_statement (
            st.dbid, st.objectid, st.encrypted,
st.text,
            er.statement_start_offset,
            er.statement_end_offset)
      as waiters_current_statement,
            qp.query_plan,
      amalgam.current_statement (
            stb.dbid, stb.objectid, stb.encrypted,
            stb.text,
            erb.statement_start_offset,
            erb.statement_end_offset)
      as blockers_current_statement,
            qp.query_plan,
      wt.*
from amalgam.dm_os_waiting_tasks_filtered wt
```

```
      left join sys.dm_exec_requests er
            on wt.waiting_task_address = er.task_address
      outer apply
sys.dm_exec_sql_text (er.sql_handle) st
            outer apply
sys.dm_exec_query_plan (er.plan_handle) qp
      left join sys.dm_exec_requests erb
            on wt.blocking_task_address =
                  erb.task_address
      outer apply
sys.dm_exec_sql_text (erb.sql_handle) stb
            outer apply
sys.dm_exec_query_plan (erb.plan_handle) qpb
```

These, however, require manual analysis of potentially verbose output. This can be eased by analyzing some of the information at the time of collection.

Often, it is useful to find the hottest resources or wait types. The following queries find all resources and wait types with at least five waiters:

```
select resource_description,
      additional_resource_description,
      count(*)
from amalgam.dm_os_waiting_tasks_filtered2
where resource_description is not null
group by resource_description, additional_resource_description
having count (*) > 5
select wait_type, count(*)
from amalgam.dm_os_waiting_tasks_filtered
group by wait_type
having count (*) > 5
```

Long waiters are generally of more concern than short-duration waiters, so it might be useful to call them out; let's see all waiters that have been waiting more than 10 seconds:

```
select *
from amalgam.dm_os_waiting_tasks_filtered
where wait_duration_ms > 10000
order by wait_duration_ms desc
```

The blocking chain also has some interesting information available (for example, head blockers that are blocking a large number of other tasks, and the chains themselves):

```
select head_blocker_task_address,
      head_blocker_session_id,
      count(*)
from amalgam.blocking_chain
group by head_blocker_task_address,
      head_blocker_session_id
having count(*) > 10
```

```
order by count(*) desc
option (maxrecursion 128)
select *
from amalgam.blocking_chain
option (maxrecursion 128)
```

The index operational statistics are also useful to have when looking for tables with high latch or lock waits:

```
select top 20 *
from sys.dm_db_index_operational_stats (
null, null, null, null)
order by page_latch_wait_count +
      page_io_latch_wait_count desc
select top 20 *
from sys.dm_exec_query_stats
order by
      (total_physical_reads +
total_logical_reads +
total_logical_writes) /
      execution_count desc
```

These are a sampling of queries that can prove useful in monitoring and then investigating blocking. Again, much of this can be easily collected using SQL Server's SQLDiag tool. Obviously, the more information that is available, the easier it is to investigate, but the costlier it is to monitor. The balance depends largely on the extra load that the system can handle without adversely affecting throughput and response times of actual application work.

Conclusion

Blocking is one of those issues that can touch many aspects of SQL Server. Although investigating blocking benefits from an understanding of how SQL Server works, it is also a good way to learn even more about the server. Of course, the immediate need to resolve blocking is often more important than learning more about the server. The intent of this chapter was to provide you with the tools and knowledge you need to face those situations when the phone is ringing off the hook on Monday morning because system performance has dropped through the floor due to heavy blocking.

And, remember that the locks in blocking you see might not have been acquired in the blocker's current statement. Look at the previous statements in the transaction; it might be immediately obvious why the locks are being held.

Other Resources

Quite a few resources deal with SQL Server blocking. Here is a sampling of some resources:

- Microsoft SQL Server Books Online. The descriptions of wait types and latch types are improving with every web release. The DMV documentation is also worth looking at.
- SQL Server Storage Engine Blog (http://blogs.msdn.com/sqlserverstorageengine/default.aspx)
- MSDN blogs in general (http://blogs.msdn.com—search for SQL and Blocking)
- *The Guru's Guide to SQL Server Architecture and Internals*, by Ken Henderson