Job Waits and iDoctor for iSeries Job Watcher White Paper

This is version 2.0 of this White Paper and contains changes specific to i5/OS V5R3.

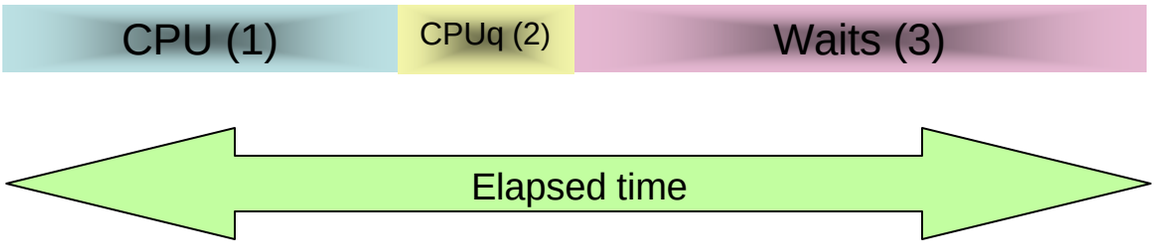
Simplistic Running and Waiting

One of the main purposes of the iDoctor for iSeries Job Watcher tool is to, in near real time, quantify the amount of "wall time" a thread/task spends "running" and the amount of time it spends "waiting". This first section introduces the concepts of running and waiting in a rather simplistic environment. The real world complexities of iSeries processor sharing, LPAR considerations, etc will be delayed until the Wait Bucket descriptions later in the paper.

All "units of work" [1] in a system at any instant in time are in one of three states:

1. "on" a CPU [2] (a.k.a. "dispatched to a processor", "using CPU", "running", "active")
2. ready to use a CPU, but waiting for a processor to become available (a.k.a. "ready", "CPU queued")
3. waiting for something or someone (a.k.a. "blocked", "idle")

A thread's *Run/Wait Signature* might look like:



How much time a unit of work spends in state 1 depends on the program design and how much work it's requested to perform as well as more complex factors such as a processor's efficiency and/or processor sharing competition (e.g. LPAR shared/partial processors).

How much time a unit of work spends in state 2 is a function of the amount of CPU competition it experiences while getting its work done.

How much time a unit of work spends in state 3 depends on many factors. But at this point we need to differentiate two types of waits:

---

[1] A "unit of work" is a single threaded job, each thread in a multi-threaded job or a system task.
[2] Hardware multithreading raises the issue of differences between the "time dispatched to a processor" and "CPU Time". More on this later.

A. Waiting for a work request to arrive (a.k.a. idle)
B. Waits that occur while performing a work request (a.k.a. blocked)

Type A waits, for example, in interactive work would be considered “key/think time”. These waits are typically not a “problem”. Or if they ARE a problem, it’s usually one external to the machine they are observed on (e.g. a communications problem causing slow arrival of work requests). Note: batch work rarely has any type A waits, unless the batch work is driven, for example, by a data queue… and the data queue is empty.

Type B waits are the interesting ones. While it’s debatable whether or not all these types of waits should be considered “problems”, the following is a safe and valid statement:

“*Outside of CPU usage and contention, type B waits are the reason jobs/threads take as long as they to do complete their work*.”

So, a more refined Run/Wait Signature for an interactive job/thread might look like:

And a typical batch type job/thread would look like:

Level set

This discussion applies to individual units of work… single threaded jobs and individual job threads. Many modern application engines involve the use of more than one job and/or more than one thread to process each “transaction”. The ideas presented in this document still apply in those cases, but each unit of work must be individually analyzed. There’s an additional burden placed on the analysis process to tie together the flow of work across the multiple jobs/threads. And, to be honest, such modern transaction engines frequently make it difficult to differentiate between type A and type B waits.

The Mysteries of Waiting

The waiting component of a job/thread’s life is easy to compute, but rarely discussed and scrutinized.

For batch type work:

```
Waits = Elapsed Time – CPU Time
```
[3]

For interactive type work:

```
Waits = Elapsed Time – CPU Time – Key/Think Time
```
[4]

What is the reason why waits have historically been ignored, unless they become so severe that the elapsed time difference becomes painfully obvious? Suggested answer: because little instrumentation or tools exist to measure and provide detail on waits. Waits are the "slightly off" relative that lives in the basement. Unless his demand for food becomes excessive, or the music gets too loud, he is best ignored. You certainly don't want to talk about him with friends.

Are waits "bad"?

This paper contends the answer is "yes". (We are obviously talking about type B waits.) There's a common misconception that a job/thread that "uses high CPU" is intrinsically bad. It MIGHT be bad. For example: If a work process normally takes 2 hours to complete with 45 minutes of CPU and, after a software or data change, now takes 4 hours with 3 hours of CPU, that IS bad. But just looking at a job/thread (in a non-comparative way) that uses a high percentage of CPU, and declaring it "bad" misses the point that "the lack or minimal occurrences of type B waits is a GOOD thing". For batch type work (that does not have type A waits, where it is waiting for work to arrive), if the type B waits are reduced/eliminated, the job/thread's "CPU Density" [5] increases. Ultimately, it could use 100% of a processor[6].

Let's take an example: A batch job that runs for 6 hours and uses 117 minutes of CPU. The first thing to consider is how much time of the "wasted" 243 minutes of elapsed time was CPU queuing (i.e. contending/waiting for a processor). This paper will go on to demonstrate how this value, and all the waits, can be measured in great detail. But for this example, let's suppose that 71 minutes of CPU queuing was involved. This means that the job was in type B waits 172 minutes. This means that the job could potentially run in 3 hours and 8 minutes… if the type B waits were completely eliminated. Contrast this with how the job might perform if the CPU speeds on the machine were doubled. One would expect the CPU minutes and CPU queuing minutes to be halved, yielding a job run

---

[3] Assumes CPU Queuing is not significant

[4] ditto 3

[5] If a single thread consumes all of a single processor for a period of time, it is 100% CPU dense. If it consumes 1/8th of a process for the same period, it is 12.5% CPU dense. This is true regardless of the number of processors on the system or in the partition. For systems with more than one CPU in the partition, CPU density is NOT what is seen on WRKACTJOB or WRKSYSACT commands. But can be computed from those, knowing how many CPUs are available to the job.

[6] DB2 Multitasking can make a job/thread appear to use more than 100% of a processor, as the background assisting tasks promote their CPU consumption numbers into the client job/thread. Note: this can also make accurate capacity planning more difficult.

time of 4.5 hours. Summary: eliminating the type B waits could have the job run in 3 hours 8 minutes. Doubling the CPU capacity could have the job run in 4 hours, 30 minutes**. Conclusion: wait analysis and reduction can be a very powerful, cost-effective way of improving response time and throughput.**

A last word on the badness of waits: An IBM "eBusiness poster" spotted outside the iSeries Benchmark Center in Rochester Minnesota contained this phrase:

All computers wait at the same speed.

Think about it.

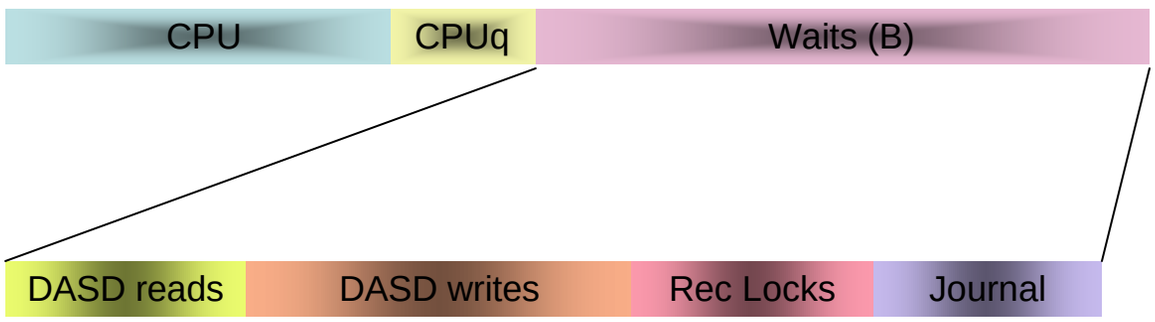
Detailing Waits

Up to here, this paper has made the case that wait analysis (and resulting "corrective actions") could lead to happiness. What is the first step in wait analysis? It begins with obtaining details on the individual waits.

Refresher: a summary Run/Wait Signature for a typical batch type job/thread might look like:

Wait analysis begins by bringing out details in the "Waits (B)" component.

For example:



This represents the first phase of detailing: the raw amount of time spent in different types of waits. The next obvious metric needed is the number of each type of wait:

Computed averages are next. Suppose the durations / counts / averages were as follows:

DASD reads	DASD writes	Rec Locks	Journal
42s	73s	45s	44s
3,523	17,772	355	5,741
0.012s	0.004s	0.126s	0.007s

This is already enough information to begin contemplating actions. Some of the questions it raises include:

How many of the DASD reads are page faults? Would main memory/pool changes help?
What objects are being read from DASD?
What programs are causing the reads?
How could those DASD reads be reduced, eliminated, or made asynchronous?
Could the DASD read response time be better?
What objects are being written to DASD?
What programs are causing the writes?
How could those DASD writes be reduce, eliminated or made asynchronous?
Could the DASD write response time be better?
What DB2 files are involved with the record locks?
What programs are requesting the record locks?
What other jobs/threads are causing the record lock contention?
What files are being journaled?
What journals are involved?
Are the journals needed and optimally configured?
Could COMMIT cycles or the Journal PRPQ used to reduce this wait component?
Is the DASD I/O subsystem write cache(s) large enough?
Is the DASD configuration well balanced, in terms of IOPs, IOAs, busses, RAID configurations?

Unfortunately it is beyond the scope of this paper to delve into details of how to tackle the wait “corrective actions”.

iDoctor for iSeries Job Watcher (JW)

All preceding material was a generic discussion of wait analysis. Now we’ll focus on such capabilities that are built into JW.

JW is a sampling based performance tool. At specified time intervals, or “as fast as possible”, a JW command/function will sample anywhere from 1 thread/job to all threads/jobs on an iSeries system. It gathers a large variety of performance data, much of it beyond the scope of this paper. But one of the main reasons for the creation of JW, was to capitalize on wait metrics first introduced into the system in Version 5 Release 1 of the Licensed Internal Code (LIC).

Remember back to the statement that a job/thread is either running on a processor, waiting for a processor to become available, or waiting for someone or something? The LIC has assigned an identifier to ALL[7] the points in LIC code that actually enter the wait state. [8] In Version 5 Release 2, there are about 165 wait points. In Version 5 Release 3, there are about 199 such wait points. Each individual wait point is sometimes referred to as an "enum". "Enum" is shorthand for the C++ programming language's "enumerated value" and simply means a fixed set of items. When a V5R3 job/thread is in the wait state, it IS in one of the 199 possible wait points. The "current wait" of a job/thread can be referred to by the numerical value of the "enum" (e.g. 51), or by a 3 character eye catcher that has been assigned to each enum (e.g. "JBw") or by a text string associated with each (e.g. "JOBUNDLEWAIT").

Wait Point Groupings, a.k.a. "Wait Buckets"

A large number of individual wait points is great from a data-empowerment point of view. However, when it comes to keeping track of them on a wait-point by wait-point basis, for every unit of work, it presents challenges to efficient implementation. An ideal design would be for each of the possible wait points (199 in V5R3) to have its own set of data associated with it, for each unit of work (job/thread/task). The minimum amount of accounting data that would be needed includes:

- Occurrence count
- Total time accumulator

It was determined that keeping 199 pairs of these numbers associated with every job/thread/task on a machine was simply too much overhead (mainly in the area of main storage footprint).

On the iSeries, a compromise was reached that allows for a potentially very large number of individual wait points to be mapped into a modest sized set of accounting data. The modest sized set of accounting numbers is called the Wait Buckets. There are 32 such buckets, but 3 of them have special purposes, so there are 29 buckets available to map the 199 wait points. Again, these buckets exist on a per unit of work basis.

Aside: Do Wait Buckets defeat the purpose of many block points?

One might ask: "What's the value in having a large number of unique block points (199, in V5R3), if all this detail is going to be lost when they get crammed into 29 Wait

---

[7] Some *types* of waits are identified with greater granularity than are other points. For example:. Locks and Seizes have more individual wait points identified than do other types of waits that tend to share block points.

[8] At the actual run/wait nitty gritty level, only LIC code can truly enter a wait. If an application or OS/400 program enters a wait state, it does so in LIC code it has caused to run on its behalf.

Buckets?" That's a fair question. The real loss of granularity is felt with sampling based tools, like Job Watcher. But even with JW, there's good use of the high wait point counts:

At any given instant in time, the full granularity afforded by all the wait points is available to sampling based tools. For example: "At this particular moment in time, thread XYZ is waiting in block point enum 114. And it has been waiting there for n microseconds."

Trace based tools, e.g. PEX Analyzer, (which are beyond the scope of this paper) can "see" every wait transition, and effectively do the accounting on a per-enum basis, making full use of the granularity provided.

For these two reasons, maximizing wait point granularity is a good thing to do.

JW Wait Points ("enums") and Wait Buckets

As mentioned earlier, wait accounting is the core functionality of the Job Watcher tool. The LIC supports **remapping** of enums to buckets. As a V5R3 iSeries system ships, not all of the 32 buckets are utilized; all enums are assigned within the first 16 buckets.[9] JW performs a remapping when it starts to utilize most of the 32 buckets, thereby maximizing the granularity of wait identification.

The bucket definitions JW uses in V5R3[10] are as follows:

1. DISPATCHED TIME
2. CPU QUEUEING TIME
3. TOTAL BLOCK TIME
4. (RESERVED)
5. DASD (PAGE FAULTS)
6. DASD (NON FAULT READS)
7. DASD SPACE USAGE CONTENTION
8. IDLE / WAITING FOR WORK
9. DASD WRITES
10. DASD (OTHER READS OR WRITES)
11. DASD OPERATION START CONTENTION
12. MUTEX/SEMAPHORE CONTENTION
13. JOURNAL SERIALIZATION

---

[9] There's a good reason for this. Collection Services also harvests Wait Bucket information, for every thread on the system. As mentioned earlier, there is a pair of numbers associated with each bucket (occurrence count, total time). By restricting the default mapping to live within the first 16 buckets, collection services reduces the number of DASD bytes required to hold the job/thread information. Keeping this amount-per-thread down is very important. Collection Services is designed so that it can be run 24 x 7 x 365 days. In that type of environment, every data item gathered on a "per-thread" basis increases the DASD space requirement for Collection Services.

[10] It can and does vary with different VRMs of the operating system.

14. MACHINE LEVEL GATE SERIALIZATION
15. SEIZE CONTENTION
16. DATABASE RECORD LOCK CONTENTION
17. OBJECT LOCK CONTENTION
18. OTHER WAITS
19. MAIN STORAGE POOL OVERCOMMITMENT
20. JAVA USER (INCLUDING LOCKS)
21. JAVA JVM
22. JAVA (OTHER)
23. SOCKET ACCEPTS
24. SOCKET TRANSMITS
25. SOCKET RECEIVES
26. SOCKET (OTHER)
27. IFS PIPE
28. IFS (OTHER)
29. DATA QUEUE RECEIVES
30. MI QUEUE (OTHER)
31. MI WAIT ON EVENTS
32. ABNORMAL CONTENTION

Additional details on the buckets and the enums that are assigned to each follow.

LIC Queuing Primitives and More Granular Wait Points

Each of the 199 block points in the system is some flavor of one of approximately 20 different LIC Queuing Primitives. Individual block points may be reported (i.e. assigned an enum) that is one of the Primitives' enums (which is the default assignment), OR (preferably) the specific block-owning LIC component can chose to "invent" another, more descriptive enum for the block point.

For example, consider synchronous DASD I/O READ wait. The author is not certain, but it is likely that the wait (block) that occurs in a job/thread while a synchronous DASD read is in progress is probably implemented with a LIC Queuing Primitive known as a "Single Task Blocker" (eye catcher QTB, enum number 4). That is, when LIC blocks a job/thread due to waiting for a synchronous DASD read to complete, it uses a QTB wait primitive/mechanism. If the component that owns this function (Storage Management) had done no further "IDing", that is how such waits would report (QTB, enum 4). That is OK, except there are probably a lot of other block points that ALSO use QTB. Therefore, it would be difficult/impossible to differentiate DASD READ blocks from other blocks. Fortunately, Storage Management, realizing how important it is to quantify DASD op waits, have invented a different eye catcher and enum (SRd, 158) that overrides QTB,4. Before you start to read this section on the Wait Buckets and their enums, you might want to read the description of Bucket 18 first. Bucket 18 contains many of the default, LIC Queuing Primitives enums.

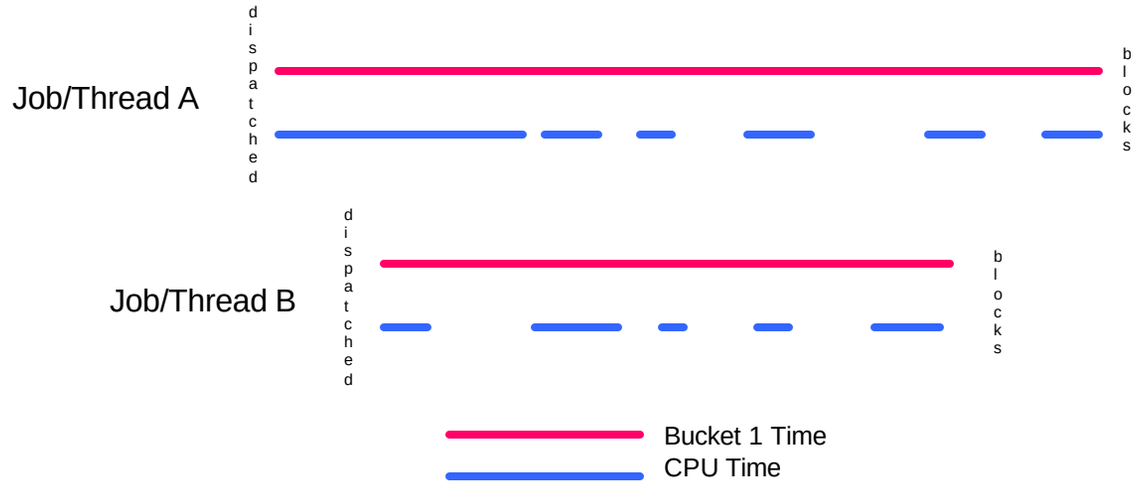
Disclaimer

The following discussion will include opinion. It will also, more than likely, be far less complete than many people (including the author) would like it to be. There's probably no single person that knows all the nuances of the 199 wait points in V5R3. Also, in spite of 199 individual points, many of these remain "general" and "generic" to some degree…preventing them from categorically being declared "normal/OK" or "bad". This discussion should be viewed as:

- Potentially in error
- Potentially out-of-date
- One person's opinion
- As a starting point, guideline to interpreting wait points and buckets, not as the "last/only word"

Bucket 1 – Dispatched Time (used to be 'CPU')

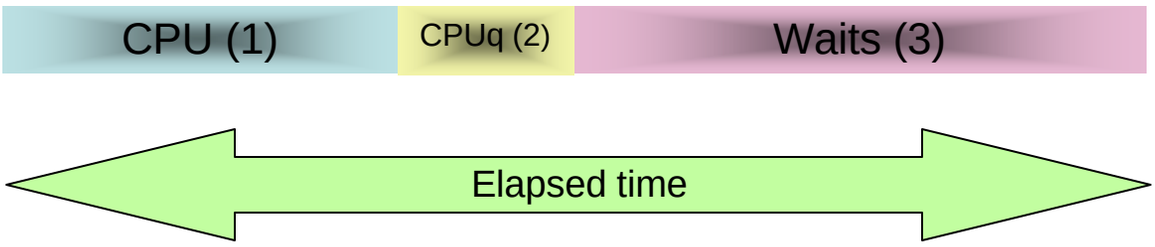
This accumulates the amount of time a thread or task has been "dispatched" to a processor. "Dispatched to a processor" means the thread or task has been assigned a processor", so it can begin execution of machine instructions. No wait points are assigned or mapped, because a dispatched thread/task is not waiting. Job Watcher, unfortunately, uses the misleading title of 'CPU' for this bucket. "CPU" is misleading because "Dispatched Time" very frequently differs from "CPU Used Time". This graphic demonstrates the relationship between "Dispatched Time" and "CPU Time".



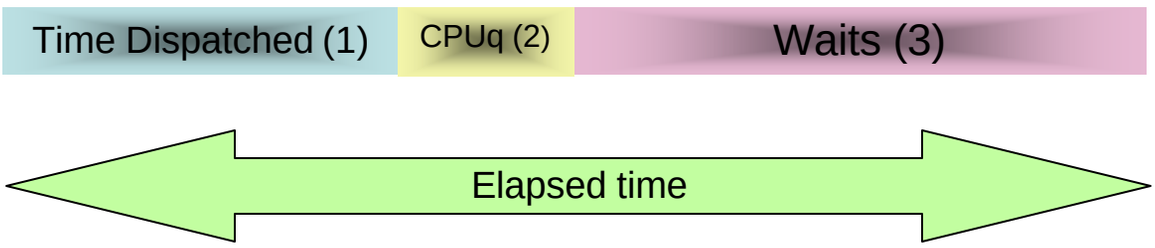
The dispatched time value can differ from the "CPU Time" measure by other means (DSPJOB, WRKACTJOB, WRKSYSACT, job accounting, the DELTACPU field in Job Watcher itself). The difference can be large. The main factors that cause these discrepancies are:

- Processor Hardware Multi Threading (HMT) feature. This can cause bucket 1's time to be larger than the actual CPU time. HMT is when more than one thread or task can be simultaneously assigned to the same physical processor. In that scenario, they share the processor's cycles, mainly during long "off chip" operations, like memory fetches. Job Watcher's Bucket 1 will record the elapsed time a thread or task has been dispatched. The real CPU value will only include the exact number of cycles used by the thread or task while it was dispatched.
- Background assisting tasks, like those used in the DB Multi-Tasking Feature. Background assisting tasks, which promote (add) their CPU usage back into the client job/thread, will cause the client thread's bucket 1 value to be smaller than the measured CPU time.[11]
- LPAR shared/partial processors. This is where the tricky concept of Virtual Processors comes into play. Bucket 1 actually records the elapsed time a thread or task is dispatched to a *Virtual Processor*, not (necessarily) a Physical Processor. Similar to HMT mentioned above, a Virtual Processor can be shared across LPAR partitions. If that occurs while a thread or task is dispatched to one of these, the bucket 1 time will be greater than the CPU time, because it will include time the thread/task is dispatched, but is "waiting for it's turn" at the physical processor behind the virtual one.

Let's revisit the **Simplistic Running and Waiting** introduction to this paper. In it, this diagram was presented:

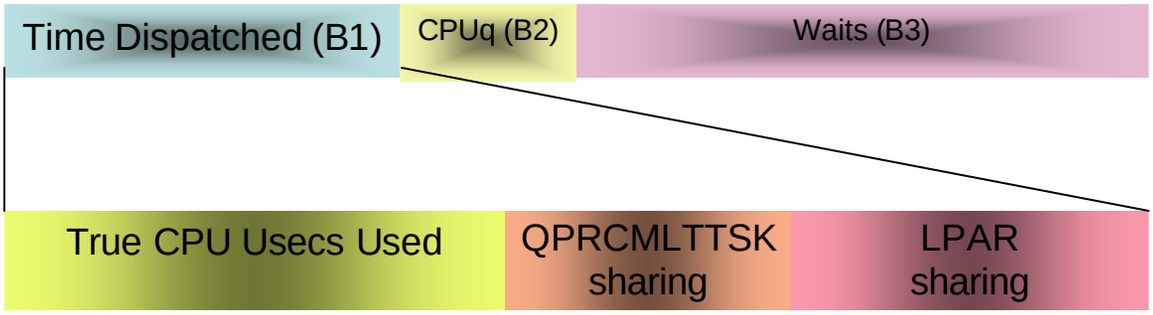


After the discussion above, the diagram should now look like:



[11] Conversely, bucket 1's value for the assisting tasks themselves would be larger (over time) than the CPU time as measured/seen by WRKSYSACT or Collection Services.

Further breakdown of (1), which we can now label **Bucket 1**, might look like:



Important:

The Job Waits Accounting Buckets used by Job Watcher do NOT break down Bucket 1 into the constituent parts shown above. However, much of the time, an additional metric gathered per interval can quantify the first "True CPU Usecs Used" part. The additional metric gathered is simply the CPU Time Used. (Again, this is a value separate from Bucket 1). When is it NOT true that this additional metric can be used to quantify the True CPU Usecs Used part of Bucket 1? Answer: if the thread being examined is having work done on its behalf by worker/assist LIC tasks. Those tasks promote/move their CPU usecs used (and several other metrics, such as DASD I/O counts, exception counts etc) back into the requesting thread's metrics. In those cases, the True CPU usecs used will be much larger than just what the thread itself had used, making it even more difficult to accurately quantify it's CPU usage. Note: worker/assist LIC tasks do NOT promote their Wait Bucket counts and times into the requesting thread… good thing!

Bucket 2 – CPU QUEUEING

No wait points assigned, waiting for a processor is a special kind of wait. This is simply the number of microseconds a thread/task has waited… ready to run… for a processor to become available.

Update:

After just having read the rather long, complex description of Bucket 1 (Time Dispatched), one learned that Time Dispatched can also in include "sharing" delays. This raises the natural question:

What's the difference between the QPRCMLTTSK/LPAR "sharing" that is part of Bucket 1 and the "queuing" that is reported in Bucket 2? After all, "queuing" and "sharing" are nearly synonymous.

The simple answer is “there is not much difference”. In particular the LPAR sharing part of Bucket 1 and all of Bucket 2 reflect wall time delays in the execution of work due to having to share a processor with other units of work. In the LPAR case, the sharing is occurring with units of work *in other partitions*. Bucket 2 time is the sharing time with other units of work *in the same partition*. QPRCMLTTSK “sharing” is somewhat more of an accounting shift, rather than true queuing/sharing.

Bucket 1 Sharing Times vs. Bucket 2 Analogy

Bucket 2 is similar to the time spent waiting in line to gain entrance to a convention hall. Bucket 1’s “sharing time” is the time spent, *after one is in side the convention*, waiting in short lines to view particular booths.

Note: even if there is little-to-no actual processor competition/sharing occurring within a partition while a thread is running, there may always be miniscule amounts of time reported in this bucket. It’s an artifact of the fact that SOME tiny, finite amount of time transpires between when a thread becomes “ready to run” and when it is dispatched to a processor.

Bucket 3 – TOTAL BLOCK TIME

This is artificial sum of all the occurrence counts and total block times for buckets 4-32.

Bucket 4 – (RESERVED)

No wait points assigned.

Bucket 5 – DASD (PAGE FAULTS)

These are the waits associated with implicit (page faults) DASD reads.

Page faults are frequently (but not exclusively) caused by having “too many jobs/threads running concurrently in too small of a main store pool”. If the faulted-on object type is a ‘1AEF’ (Temporary Process Control Space), then that is a likely cause. There are other types of activity, though, where page faults are expected or “normal”:

- When a program or application first starts up in a job/thread.
- DB2 Access Paths (keyed parts of physical files, or Logical Files)… these tend to be referenced in a highly unpredictable way, and “faulting in” pages of access paths is considered “normal”.

The enums associated with this bucket are:

Eye

```
Enum  Catcher  Description
161     SFt    MAINSTORE/LOGICAL-DASD-IO: PAGE FAULT
162     SFP    MAINSTORE/LOGICAL-DASD-IO: PAGE FAULT IO PENDING
164     GRf    MAINSTORE/LOGICAL-DASD-IO: ACCESS GROUP READ FOR FAULT
165     SRR    MAINSTORE/LOGICAL-DASD-IO: ACCESS GROUP READ FOR FAULT IO
PENDING
```

Bucket 6 – DASD (NON FAULT READS)

These are simply the waits associated with explicit ("read this from DASD for me") **synchronous** DASD reads.

The enums associated with this bucket are:

```
        Eye
Enum  Catcher  Description
158     SRd    MAINSTORE/LOGICAL-DASD-IO: DASD READ
159     SRQ    MAINSTORE/LOGICAL-DASD-IO: DASD READ IO PENDING
```

Bucket 7 – DASD SPACE USAGE CONTENTION

When an object, or internal LIC object is created or extended, and free DASD space has to be located to satisfy the request, there is some level of serialization performed. This is done on an ASP-by ASP and unit-by-unit basis. Normally, one would expect to see little, if any, of these types of waits. If they are present in significant percentages, it usually means the OS/LIC is being asked (by applications) to perform a very high RATE of object creates/extends/truncates or deletes. (Note: Opening a DB2 file causes a create.) The SIZE of the DASD space requests is not relevant to these blocks; it's the RATE of requests that is relevant.

The enums associated with this bucket are:

```
        Eye
Enum  Catcher  Description
145     ASM    DASD SPACE MANAGER: CONCURRENCY CONTENTION
146     ASM    DASD SPACE MANAGER: ASP FREE SPACE DIRECTORY
147     ASM    DASD SPACE MANAGER: RR FREE SPACE LOCK
148     ASM    DASD SPACE MANAGER: GP FREE SPACE LOCK
149     ASM    DASD SPACE MANAGER: PERMANENT DIRECTORY LOCK
180     ASM    DASD SPACE MANAGER: TEMPORARY DIRECTORY LOCK
181     ASM    DASD SPACE MANAGER: PERSISTENT STORAGE LOCK
182     ASM    DASD SPACE MANAGER: STATIC DIRECTORY LOCK
183     ASM    VIRTUAL ADDRESS MANAGER: BIG SEGMENT ID LOCK
184     ASM    VIRTUAL ADDRESS MANAGER: LITTLE SEGMENT ID LOCK
185     ASM    DASD SPACE MANAGER: IASP LOCK
186     ASM    DASD SPACE MANAGER: MOVE CHAIN
187     ASM    DASD SPACE MANAGER: HYPERSPACE LOCK
188     ASM    DASD SPACE MANAGER: NON PERSISTENT DATA LOCK
189     ASM    VIRTUAL ADDRESS MANAGER: TEMPORARY SEGMENT ID RANGE MAPPER LOCK
190     ASM    VIRTUAL ADDRESS MANAGER: PERMANENT SEGMENT ID RANGE MAPPER LOCK
191     ASM    VIRTUAL ADDRESS MANAGER: IASP SEGMENT ID RANGE MAPPER LOCK
```

Bucket 8 – IDLE / WAITING FOR WORK

These are the waits on MI queue associated with each OS job known as the "MI Response Queue". Normally, for 5250 type interactive applications, this would reflect the key/think time. Other possible uses would be APPC/APPN SNA type communications waits.

The enums associated with this bucket are:

Enum	Eye Catcher	Description
340	QMr	IDLE WAIT, MI RESPONSE QUEUE WAIT

Bucket 9 – DASD WRITES

These are the waits associated with **synchronous** DASD writes, **or waiting for asynchronous DASD writes to complete**.

The enums associated with this bucket are:

Enum	Eye Catcher	Description
167	SWt	MAINSTORE/LOGICAL-DASD-IO: DASD WRITE
168	SWP	MAINSTORE/LOGICAL-DASD-IO: DASD WRITE IO PENDING
170	SWp	MAINSTORE/LOGICAL-DASD-IO: PAGE OUT WRITE
171	GPg	MAINSTORE/LOGICAL-DASD-IO: ACCESS GROUP PURGE
172	GPP	MAINSTORE/LOGICAL-DASD-IO: ACCESS GROUP PURGE IO PENDING
174	GTA	MAINSTORE/LOGICAL-DASD-IO: GENERIC ASYNC IO TRACKER WAIT
175	GTS	MAINSTORE/LOGICAL-DASD-IO: GENERIC SINGLE TASK BLOCKER WAIT
176	GTT	MAINSTORE/LOGICAL-DASD-IO: GENERIC TIMED TASK BLOCKER

Bucket 10 – DASD (OTHER READS OR WRITES)

The enums with the 'DSM' eye-catcher deal primarily with actions taken to do DASD unit configuration and setup and should rarely be seen in "production jobs/threads".

The other enums with an eye-catcher other than 'DSM' are DASD op waits that can't be differentiated by read or write type of operations. These should rarely occur.

The enums associated with this bucket are:

Enum	Eye Catcher	Description
60	DSM	DASD MANAGEMENT OPS: FIND COMPRESSION GROUP
61	DSM	DASD MANAGEMENT OPS: DEALLOCATE COMPRESS GROUP
62	DSM	DASD MANAGEMENT OPS: READ COMPRESSION DIRECTORY
63	DSM	DASD MANAGEMENT OPS: WRITE COMPRESSION DIRECTORY

```
64      DSM    DASD MANAGEMENT OPS: INIT COMPRESSION START REORG
65      DSM    DASD MANAGEMENT OPS: MIRROR READ SYNC
66      DSM    DASD MANAGEMENT OPS: MIRROR REASSIGN SYNC
67      DSM    DASD MANAGEMENT OPS: MIRROR WRITE VERIFY SYNC
68      DSM    DASD MANAGEMENT OPS: READ
69      DSM    DASD MANAGEMENT OPS: READ DIAG
70      DSM    DASD MANAGEMENT OPS: VERIFY
71      DSM    DASD MANAGEMENT OPS: VERIFY DIAG
72      DSM    DASD MANAGEMENT OPS: WRITE
73      DSM    DASD MANAGEMENT OPS: WRITE DIAG
74      DSM    DASD MANAGEMENT OPS: WRITE VERIFY
75      DSM    DASD MANAGEMENT OPS: WRITE VERIFY DIAG
76      DSM    DASD MANAGEMENT OPS: REASSIGN
77      DSM    DASD MANAGEMENT OPS: REASSIGN DIAG
78      DSM    DASD MANAGEMENT OPS: ALLOCATE
79      DSM    DASD MANAGEMENT OPS: ALLOCATE DIAG
80      DSM    DASD MANAGEMENT OPS: DEALLOCATE
81      DSM    DASD MANAGEMENT OPS: DEALLOCATE DIAG
82      DSM    DASD MANAGEMENT OPS: ENABLE AUTO ALLOCATE
83      DSM    DASD MANAGEMENT OPS: DISABLE AUTO ALLOCATE
84      DSM    DASD MANAGEMENT OPS: QUERY COMPRESSION METRICS
85      DSM    DASD MANAGEMENT OPS: QUERY COMPRESSION METRICS DIAG
86      DSM    DASD MANAGEMENT OPS: COMPRESSION SCAN READ
87      DSM    DASD MANAGEMENT OPS: COMPRESSION SCAN READ DIAG
88      DSM    DASD MANAGEMENT OPS: COMPRESSION DISCARD TEMP DATA
89      DSM    DASD MANAGEMENT OPS: COMPRESSION DISCARD TEMP DATA DIAG
150     STv    MAINSTORE/LOGICAL-DASD-IO: SAR NOT SET
151     SRv    MAINSTORE/LOGICAL-DASD-IO: REMOVE
152     SRP    MAINSTORE/LOGICAL-DASD-IO: REMOVE IO PENDING
153     SCl    MAINSTORE/LOGICAL-DASD-IO: CLEAR
154     SCP    MAINSTORE/LOGICAL-DASD-IO: CLEAR IO PENDING
156     SUp    MAINSTORE/LOGICAL-DASD-IO: UNPIN
157     SUP    MAINSTORE/LOGICAL-DASD-IO: UNPIN IO PENDING
177     SMP    MAINSTORE/LOGICAL-DASD-IO: POOL CONFIGURATION
178     SMC    MAINSTORE/LOGICAL-DASD-IO: POOL CONFIGURATION CHANGE
```

Bucket 11 – DASD OPERATION START CONTENTION

These waits occur when a DASD operation start is delayed due to a very high rate of concurrent DASD operations in progress at the moment it is requested.

The enums associated with this bucket are:

```
        Eye
Enum  Catcher  Description
49      QRR    QURESSTACKMSGPOOL, ABNORMAL DASD OP START CONTENTION
```

Bucket 12 – MUTEX/SEMAPHORE CONTENTION

These are the block points used by C/C++ programming language (both operating system code, LPP and application code), usually in the POSIX environment, to implement Mutex and Semaphore waits.

The enums associated with this bucket are:

Enum	Eye Catcher	Description
15	QMG	QUMUTEXGATE, NOT OTHERWISE IDENTIFIED
16	QSm	QUSEMAPHORE, NOT OTHERWISE IDENTIFIED

Bucket 13 - JOURNAL SERIALIZATION

The waits associated with DB2 Journaling are in this bucket.

The enums associated with this bucket are:

Enum	Eye Catcher	Description
50	JBo	JOURNAL BUNDLE OWNER WAIT FOR DASD COMPLETION
51	JBw	JOURNAL BUNDLE WAIT FOR DASD COMPLETION
260	EFJ	EPFS: WAIT FOR OS TO FINISH APPLY JOURNAL CHANGES
261	ERJ	EPFS: WAIT FOR OS REQUEST TO APPLY JOURNAL CHANGES

Enum 50 is the wait in the thread that is actually performing the DASD write(s) to the journal. It is the wait for DASD journal writes to complete. Journal uses some fancy approaches to DASD ops to do their writes absolutely as efficiently as possible. That is why DASD writes to journals do not fall in the “DASD Write” bucket below (this is a good thing for performance analysis, to have these journal writes differentiated).

Enum 51 is the wait that occurs in threads other than the one that’s performing the DASD write(s). For efficiency, multiple jobs/threads can “ride along” the journal DASD writes performed by other jobs/threads.

Bucket 14 - MACHINE LEVEL GATE SERIALIZATION

The enums associated with this bucket are:

Enum	Eye Catcher	Description
2	QGa	QUGATE, NOT OTHERWISE IDENTIFIED
3	QTG	QUTRYGATE, NOT OTHERWISE IDENTIFIED

QGa is a very high performance, low-overhead serialization primitive used by LIC. It is the type of primitive in which there can be one and only one “holder”. Normally, QGa is used in areas in which the anticipated wait time, if any, is very small (microseconds). Note: there are some related block points (QGb, QGc, QGd) that are later covered in the bucket named “ABNORMAL CONTENTION”.

Bucket 15 - SEIZE CONTENTION

Think of seizes as the Licensed Internal Code's (LIC's) equivalent of Locks. A seize almost always occurs on/against an MI object (DB2 physical file member, Data Queue, Program, Library…). Seizes can conflict with Locks and can cause Lock conflicts. There is a large variety of seizes: shared, exclusive, "fair", and "intent-exclusive". It's beyond the scope of this paper to explain all there is to know about seizes. They are, after all, internal LIC primitives that are subject to change at any time. If seizes are a significant percentage of a Run/Wait Signature, examining the call stack, "wait object" and "holding task/thread" (if any) are probably necessary to understand what is causing the contention.

Seizes are frequently (but not exclusively) associated with data base objects and operations. Concurrent activities in multiple jobs such as opens, closes, journal sync points, access path building, etc might lead to seize waits. Other actions/objects that can experience seize waits include libraries and user profiles, during high rates of concurrent Create/Delete activity in multiple jobs.

This bucket was the first time that the term "holding task/thread" was mentioned. However, Job Watcher has that ability to determine the "holder" for more than just seize waits. It can do so for Locks, Data Base Record Locks and other wait enums based on a low level serialization primitive called a "gate".

In the area of waiters and holders, it needs to be pointed out that the waiter… the job/thread that is experiencing the wait… is frequently the victim, not the culprit.

The enums associated with this bucket are:

Enum	Eye Catcher	Description
100	Rex	SEIZE: EXCLUSIVE
101	Rex	SEIZE: LONG RUNNING EXCLUSIVE
102	Rsh	SEIZE: SHARED
103	Rix	SEIZE: INTENT EXCLUSIVE
104	Ris	SEIZE: INTENT SHARED
105	Rfa	SEIZE: FLUSH ALL
106	Rdx	SEIZE: DATABASE EXCLUSIVE
107	Rii	SEIZE: INTERNAL INTENT EXCLUSIVE
108	Rot	SEIZE: OTHER
109	Rlk	SEIZE: LOCK CONFLICT
112	RXX	SEIZE/LOCK IMPOSSIBLE
125	Rsp	SEIZE: OFF-LINE IASP
126	Rra	SEIZE: RELEASE ALL
127	Rrs	SEIZE: RELEASE
133	Rss	SEIZE/LOCK: INTERNAL SERVICE TOOLS HASH CLASS GATE
135	Rmf	SEIZE: MONITORED FREE
141	Rcu	SEIZE: CLEANUP
143	Rsv	SEIZE/LOCK: SERVICE
320	SOo	COMMON MI OBJECT CHECKER: SEIZE OBJECT
321	SOi	COMMON MI OBJECT CHECKER: SEIZE FOR IPL NUMBER CHECK

Bucket 16 - DATABASE RECORD LOCK CONTENTION

Hopefully these enums are self-explanatory.

The enums associated with this bucket are:

Enum	Eye Catcher	Description
110	RDr	DB RECORD LOCK: READ
111	RDu	DB RECORD LOCK: UPDATE
123	RDw	DB RECORD LOCK: WEAK
134	Rxf	DB RECORD LOCK: TRANSFER
136	Rck	DB RECORD LOCK: CHECK
139	Rcx	DB RECORD LOCK: CONFLICT EXIT

Bucket 17 - OBJECT LOCK CONTENTION

These are the conflicts between threads involving objects. The OS frequently needs/obtains locks during such operations as:

- Opening a DB2 file
- Creating/deleting an object into a library
- Moving an object to a different library
- Ownership changes

The operating system can also use “symbolic locks” as a serialization mechanism. These are called “space location locks”.

Lastly, application code can explicitly use locks via the ALCOBJ CL command.

The enums associated with this bucket are:

Enum	Eye Catcher	Description
113	RIr	LOCK: SHARED READ
114	RIo	LOCK: SHARED READ ONLY
115	RIu	LOCK: SHARED UPDATE
116	RIa	LOCK: EXCLUSIVE ALLOW READ
117	RIe	LOCK: EXCLUSIVE NO READ
118	RMr	LOCK: SEIZE CONFLICT, EXCLUSIVE
119	RMo	LOCK: SEIZE CONFLICT, SHARED
120	RMu	LOCK: SEIZE CONFLICT, INTENT EXCLUSIVE
121	RMa	LOCK: SEIZE CONFLICT, INTENT SHARED
122	RMe	LOCK: SEIZE CONFLICT, INTERNAL INTENT EXCLUSIVE
124	RMm	LOCK: MATERIALIZE
128	Rdo	LOCK: DESTROY OBJECT
129	Rdp	LOCK: DESTROY PROCESS
130	Rdt	LOCK: DESTROY THREAD
131	Rdx	LOCK: DESTROY TRXM
132	Rar	LOCK: ASYNC RETRY
137	Rtr	LOCK: TRACE
138	Rul	LOCK: UNLOCK
140	Rlc	LOCK: LOCK COUNT
142	Rpi	LOCK: PROCESS INTERRUPT

Note: the enums with the word “SEIZE” in the description are lock conflicts caused by existing seizes on an object.

Bucket 18 - OTHER WAITS

The dreaded “other” word! Yes, even JW’s wait accounting has to have a “catch all bucket”.

The enums associated with this bucket are:

Enum	Eye Catcher	Description
1	QCo	QUCOUNTER, NOT OTHERWISE IDENTIFIED
4	QTB	QUSINGLETASKBLOCKER, NOT OTHERWISE IDENTIFIED
5	QUW	QUUNBLOCKWHENDONE, NOT OTHERWISE IDENTIFIED
6	QQu	QUQUEUE, NOT OTHERWISE IDENTIFIED
7	QTQ	QUTREEQUEUE, NOT OTHERWISE IDENTIFIED
9	QPo	QUPOOL, NOT OTHERWISE IDENTIFIED
10	QMP	QUMESSAGEPOOL, NOT OTHERWISE IDENTIFIED
11	QMP	QUSIMPLEMSGPOOL, NOT OTHERWISE IDENTIFIED
12	QSP	QUSTACKLESSMSGPOOL, NOT OTHERWISE IDENTIFIED
13	QSC	QUSTATECOUNTER, NOT OTHERWISE IDENTIFIED
17	QSB	QUSYSTEMBLOCKER, NOT OTHERWISE IDENTIFIED
240	RCA	LIC CHAIN FUNCTIONS: SMART CHAIN ACCESS
241	RCI	LIC CHAIN FUNCTIONS: SMART CHAIN ITERATOR
242	RCM	LIC CHAIN FUNCTIONS: CHAIN MUTATOR
243	RCB	LIC CHAIN FUNCTIONS: SMART CHAIN PRIORITY BUMP 1
244	RCB	LIC CHAIN FUNCTIONS: SMART CHAIN PRIORITY BUMP 2
245	RCE	LIC CHAIN FUNCTIONS: CHAIN ACCESS EXTENDED

The above enums with eye catchers beginning with a ‘Q’ are the generic wait points… the low level LIC blocks that have not (yet) been uniquely identified. These enums will be seen when LIC code blocks that has not gone out of its way to uniquely identify the block point. The only identification that exists is the differentiation afforded by the type of LIC blocking primitive used. A few words/opinions can be offered for some of them:

QCo is frequently used for timed waits. The wait used at the core of the DLY JOB command is a QCo wait. It is also used by POSIX Condition Variable waits.

QTB is a wait primitive used for many purposes (unfortunately). About the only generic statement that can be made on it is that is used when a thread/task is waiting for a specific action to happen on its behalf… explicitly for THAT thread/task. For example, waiting for synchronous DASD reads and writes to complete use QTB blocks. Fortunately, DASD reads and writes have further been identified, so they are covered by their own unique buckets, they are not lumped into QTB (see other buckets).

Bucket 19 - MAIN STORAGE POOL OVERCOMMITMENT

These waits indicate one or more main storage pools are currently overcommitted. Regular operations, like explicit DASD reads or page faults, are being delayed in order to locate “free” main storage page frames to hold the new incoming data.

The enums associated with this bucket are:

```
        Eye
Enum  Catcher  Description
155     GCP    MAINSTORE/LOGICAL-DASD-IO: CLEAR PAGE OUT WAIT
160     GRQ    MAINSTORE/LOGICAL-DASD-IO: DASD READ PAGE OUT WAIT
163     GFP    MAINSTORE/LOGICAL-DASD-IO: PAGE FAULT PAGE OUT WAIT
166     GRR    MAINSTORE/LOGICAL-DASD-IO: ACCESS GROUP READ FOR FAULT PAGE OUT
169     GWP    MAINSTORE/LOGICAL-DASD-IO: DASD WRITE PAGE OUT WAIT
173     SPw    MAINSTORE/LOGICAL-DASD-IO: PAGE OUT WAIT, POOL OVERCOMMITMENT
```

Bucket 20 - JAVA USER (INCLUDING LOCKS)

The enums associated with this bucket are:

```
        Eye
Enum  Catcher  Description
200     JUW    JAVA: USER WAIT
201     JSL    JAVA: USER SLEEP
203     JSU    JAVA: SUSPEND WAIT
209     JOL    JAVA: OBJECT LOCK
304     JSG    JAVA: SYNCHRONOUS GARBAGE COLLECTOR WAIT
305     JSF    JAVA: SYNCHRONOUS FINALIZATION WAIT
```

Bucket 21 – JAVA JVM

The enums associated with this bucket are:

```
        Eye
Enum  Catcher  Description
302     JWH    JAVA: GARBAGE COLLECTOR WAIT HANDSHAKE WAIT
303     JPH    JAVA: PRIMARY GC THREAD WAIT FOR HELPER THREADS DURING SWEEP
306     JGW    JAVA: GARBAGE COLLECTOR WAITING FOR WORK
307     JFW    JAVA: FINALIZATION WAITING FOR WORK
308     JVW    JAVA: VERBOSE WAITING FOR WORK
```

Bucket 22 - JAVA (OTHER)

The enums associated with this bucket are:

```
        Eye
Enum  Catcher  Description
202     JWC    JAVA: WAIT FOR COUNT
204     JEA    JAVA: END ALL THREADS
205     JDE    JAVA: DESTROY WAIT
206     JSD    JAVA: SHUTDOWN
207     JCL    JAVA: CLASS LOAD WAIT
208     JSL    JAVA: SIMPLE LOCK
300     JGG    JAVA: GARBAGE COLLECTOR GATE GUARD WAIT
301     JAB    JAVA: GARBAGE COLLECTOR ABORT WAIT
309     JGD    JAVA: GARBAGE COLLECTION DISABLE WAIT
```

```
310     JGE    JAVA: GARBAGE COLLECTION ENABLE WAIT
```

Bucket 23 - SOCKET ACCEPTS

These are socket op block points associated with the socket accept() API call. Normally, but not always, these represent a thread “waiting for work”.

The enums associated with this bucket are:

```
        Eye
Enum  Catcher  Description
210     STA    COMM/SOCKETS: SHORT WAIT FOR ACCEPT
211     LTA    COMM/SOCKETS: LONG WAIT FOR ACCEPT
```

Bucket 24 - SOCKET TRANSMITS

These are waits associated with Socket APIs calls that are sending/transmitting data.

The enums associated with this bucket are:

```
        Eye
Enum  Catcher  Description
212     STS    COMM/SOCKETS: SHORT WAIT FOR TCP SEND
213     LTS    COMM/SOCKETS: LONG WAIT FOR TCP SEND
216     SUS    COMM/SOCKETS: SHORT WAIT FOR UDP SEND
217     LUS    COMM/SOCKETS: LONG WAIT FOR UDP SEND
```

Bucket 25 - SOCKET RECEIVES

These are waits associated with Socket APIs calls that are receiving data.

The enums associated with this bucket are:

```
        Eye
Enum  Catcher  Description
214     STR    COMM/SOCKETS: SHORT WAIT FOR TCP RECEIVE
215     LTR    COMM/SOCKETS: LONG WAIT FOR TCP RECEIVE
218     SUR    COMM/SOCKETS: SHORT WAIT FOR UDP RECEIVE
219     LUR    COMM/SOCKETS: LONG WAIT FOR UDP RECEIVE
```

Bucket 26 - SOCKET (OTHER)

The primary wait points that should be seen from this bucket involve the SELECT socket API. That API can be used by an application for a variety of complex waiting scenarios.

The enums associated with this bucket are:

```
        Eye
Enum  Catcher  Description
220     SAS    COMM/SOCKETS: SHORT WAIT FOR IO COMPLETION
221     LAS    COMM/SOCKETS: LONG WAIT FOR IO COMPLETION
222     SSW    COMM/SOCKETS: SELECT SHORT WAIT
223     SLW    COMM/SOCKETS: SELECT LONG WAIT
```

Bucket 27 - IFS PIPE

These waits are due to Integrated File System (IFS) “pipe” operations.

The enums associated with this bucket are:

```
        Eye
Enum  Catcher  Description
252     PPC    IFS/PIPE: MAIN PIPE COUNT
253     PRP    IFS/PIPE: READ END OF PIPE
254     PWP    IFS/PIPE: WRITE END OF PIPE
255     PRW    IFS/PIPE: PIPE READ WAITERS
256     PWW    IFS/PIPE: PIPE WRITE WAITERS
```

Bucket 28 - IFS (OTHER)

Hopefully, the descriptions on these IFS blocks points need no further elaborations.

The enums associated with this bucket are:

```
        Eye
Enum  Catcher  Description
250     PRL    IFS/PIPE: FILE TABLE ENTRY EXCLUSIVE LOCK
251     PRC    IFS/PIPE: LIC REFERENCE COUNT
```

Bucket 29 - DATA QUEUE RECEIVES

These are the waits on MI Data Queue objects.

The enums associated with this bucket are:

```
        Eye
Enum  Catcher  Description
341     QMd    DATA QUEUE WAIT
```

Bucket 30 - MI QUEUE (OTHER)

These are waits on MI Queue objects other than the two preceding types. In general, these would be internal OS operations[12] or User Queue dequeue waits.

The enums associated with this bucket are:

```
        Eye
Enum  Catcher  Description
342     QMo    OTHER MI QUEUE WAIT
```

Bucket 31 - MI WAIT ON EVENTS

Event waits are used mainly across jobs, by internal OS programs.

The enums associated with this bucket are:

```
        Eye
Enum  Catcher  Description
330     EMw    MI EVENT WAIT
```

Bucket 32 - ABNORMAL CONTENTION

These waits reflect a high rate of concurrent waits/releases occurring against a wide variety of many of the other wait points listed previously. There are two types of these waits:

a. Unsuccessful wakeup retries (QGb, QGc, QGd)
b. Waiting in line to buy a ticket that gets you into the main wait line (QWL)

The enums associated with this bucket are:

```
        Eye
Enum  Catcher  Description
8       QRP    QURESSTACKMSGPOOL, NOT OTHERWISE IDENTIFIED
14      QWL    QUWAITLIST, WAITING FOR ACCESS TO A WAIT LIST
40      QGb    QUGATEB, ABNORMAL QUGATE CONTENTION, FIRST RETRY
41      QGc    QUGATEC, ABNORMAL QUGATE CONTENTION, SECOND RETRY
42      QGd    QUGATED, ABNORMAL QUGATE CONTENTION, THIRD RETRY
```

[12] For example, most subsystem monitor jobs' normal wait point is a dequeue on an MI queue (that is neither the MI Response Queue nor a Data Queue).

Trademarks and Disclaimers

References in this document to IBM products or services do not imply that IBM intends to make them available in every country.

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

AS/400	e-business on demand	OS/400
IBM	IBM (logo)	iSeries
eServer	iDoctor for iSeries	

MMX, Pentium, and ProShare are trademarks or registered trademarks of Intel Corporation in the United States, other countries, or both.
Microsoft and Windows NT are registered trademarks of Microsoft Corporation in the United States, other countries, or both.
Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.
SET and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.
C-bus is a trademark of Corollary, Inc. in the United States, other countries, or both.
UNIX is a registered trademark of The Open Group in the United States and other countries.
Other company, product or service names may be trademarks or service marks of others.

Information is provided "AS IS" without warranty of any kind.

All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics may vary by customer.

Information concerning non-IBM products was obtained from a supplier of these products, published announcement material, or other publicly available sources and does not constitute an endorsement of such products by IBM. Sources for non-IBM list prices and performance numbers are taken from publicly available information, including vendor announcements and vendor worldwide homepages. IBM has not tested these products and cannot confirm the accuracy of performance, capability, or any other claims related to non-IBM products. Questions on the capability of non-IBM products should be addressed to the supplier of those products.

All statements regarding IBM future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. Contact your local IBM office or IBM authorized reseller for the full text of the specific Statement of Direction.

Some information addresses anticipated future capabilities. Such information is not intended as a definitive statement of a commitment to specific levels of performance, function or delivery schedules with respect to any future products. Such commitments are only made in IBM product announcements. The information is presented here to communicate IBM's current investment and development activities as a good faith effort to help with our customers' future planning.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Photographs shown are of engineering prototypes. Changes may be incorporated in production models.