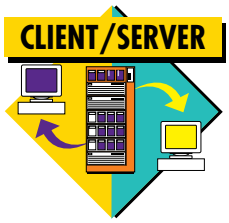


SMP Performance

By Bret R. Olszewski and Jim Van Fleet

The rapidly increasing population of AIX-based Symmetric Multiprocessing (SMP) systems has sparked an interest in understanding the performance characteristics of these systems. Although the concepts important to performance on uniprocessor systems are the same, SMP performance does have some unique characteristics. This article introduces the performance problem discovery and resolution on RISC System/6000® SMPs.

Scaling is the performance increase associated with adding more processors to a system. It also represents the fundamental performance difference between uniprocessor and Symmetric Multiprocessor (SMP) systems.[1,2]

Three basic components of system scaling are hardware, system software, and application software. Hardware factors include bandwidth and latency within the system. Latency defines the maximum performance of a system, since the number of instructions executed is related to the amount of time the processor spends waiting on resources such as cache, memory, and I/O. Bandwidth is an important hardware component because it defines scalability. As processors begin to contend for resources, delays occur, which reduces overall system performance.

System software (primarily operating systems) scaling is primarily paced by serialization (locking) effects. Most software uses locks to guard access to critical data structures. For example, when you transfer money from your bank savings account to your checking account, a lock is logically taken to ensure that other transactions occurring at the same time do not impact the resulting balances on your account. Application software frequently has similar characteristics, such as locks, to system software in serialization and synchronization.

In system scaling for real applications, performance does not scale linearly with the number of processors.[3] For example, doubling the number of processors does not double the throughput of the system. In addition, not all systems or workloads scale similarly; in fact, quite the opposite is true. Particularly in hardware, scaling potential is more often related to system cost; low-cost systems are optimized for cost, not performance. Systems optimized for performance generally have a higher cost.

Tools

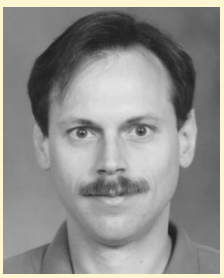
AIX provides a wide variety of performance tools for evaluating system performance. Although this article does not detail the features of performance tools, it does review the basic capabilities of some useful tools. Figure 1 describes several tools and their functions.

SMP Performance Issues

The most important categories of SMP performance problems are described below.

Workload Imbalance

Workload imbalance occurs when the workload has insufficient parallelism to keep multiple processors busy. For example, if a large FORTRAN program is run without any parallelization on an SMP, it will complete only as fast as one processor can run it; the other processors in the system will have nothing to do. Typically, any system will have some non-parallel tasks (such as some backup utilities) that may become performance bottlenecks.



Bret R. Olszewski

Jim Van Fleet

[1]Blakely-Fogel, Debora and Alexander,William. "SMP Overview." *AIXpert* (November 1994) p. 4.
[2]Alexander, William; Dimpsey, Robert; and Olszewski, Bret R. "AIX Operating System SMP Performance." *AIXpert* (November 1994) p. 35.
[3]Dixit, K.; Van Fleet, J.; Olszewski, B. "Workload Effects on SMP Scaling in AIX Version 4." Compcon '96 Proceedings. p. 117.

Another possible imbalance could be caused by funneled device drivers. *Funneling* is an AIX feature that allows device drivers not optimized for MP to run correctly. This occurs only by allowing the device driver to execute on one CPU (always CPU 0) in the system. (CPU balance can be observed with `sar`.) Although few device drivers shipped with AIX are funneled, that possibility exists for any driver, particularly third-party supplied drivers. The signature of workload imbalance is lower-than-expected CPU utilization.

Application Serialization

Application serialization manifests itself when threads must protect common resources, usually when using AIX kernel services. Normally, the characteristic of this situation is high use of system calls, particularly those associated with semaphores, file locks, message queues, or pipes. Application serialization is often noted for its high kernel CPU utilization.

Thread Serialization

Thread serialization is similar to application serialization, but it happens only when using POSIX™ threads library services. Because the threads library contains much code that runs in the user space, the characteristics of thread serialization are more subtle than application serialization. Thread serialization generally has lower-than-expected CPU utilization.

Kernel Locking Conflict

Kernel locking conflicts occur when either one or a few AIX locks are heavily used as the result of a workload. Disabled AIX locks function as spin locks; that is, when one thread attempts to access a lock held by another thread, it will repeatedly attempt to obtain the lock (spin). In other cases, the thread attempting to obtain the lock may voluntarily give up the processor, which causes a context switch to another thread.

The signature of kernel locking conflicts is high kernel CPU utilization and often high dispatch rates. High dispatch rates, measured on 8-way PowerPC 601® systems on tuned workloads, often range from 720 context switches per second to 4400 switches per second. Obviously, reasonable rates will be lower on systems with fewer processors.

MP/UP Overhead

AIX provides two kernels: a uniprocessor kernel that runs only on uniprocessor systems and an MP kernel that runs on SMP systems. This dual kernel approach allows AIX to optimize some functions for uniprocessor systems.

MP/UP overhead is the path-length difference between the two kernels. The performance difference for the two kernels can be quantified by running the MP kernel on a UP system. Normally, the MP/UP overhead of AIX is between 5% and 10%, but it depends on how much time the workload spends in the AIX kernel. The effect of MP/UP overhead is generally an issue only when comparing 2-way systems to uniprocessor systems on workloads that contain a large amount of kernel time.

Tool	Function
`-iostat`	Displays CPU and disk utilization of the system; divides processor utilization into time spent in kernel, user, idle, and disk I/O wait.
`-vmstat`	Displays CPU utilization, paging rate, and context switch rate, as well as memory pool utilization; reports processor utilization similar to `iostat`.
`-sar`	Displays CPU utilization and many other operating system counts; can display CPU load balance (collect `sar` data as `/usr/lib/sa/sadc 10 5 sardata.bin`, then process with `sar -P ALL -f sardata.bin`). Although CPU load balance is typically not a problem, it can be a useful metric.
`-lockstat`	Displays lock contention in the AIX kernel and kernel extensions; it requires that the system be booted with lock instrumentation enabled (reference `bosboot -L` command). The overhead of enabling lock instrumentation is typically 3% to 5%. Normally, only locks that have more than 1,000 references a second will have much impact on performance. Performance problems caused by AIX locking are uncommon.
`-tprof`	Samples CPU utilization to account time in threads and program modules. Although it is not designed specifically to identify SMP issues, it can help identify scaling problems in applications.
`-trace`	Collects time-stamped kernel events (see "utld: A Trace-based Performance Tool" in this issue). Trace has higher system overhead than most tools. *Hint:* when examining printable output via `trcrpt` for SMPs, use the `-O cpuid=on` flag to show the processor number for each event.

Figure 1. Tools for evaluating system performance

Hardware Configuration

Bandwidth limits can cause hardware configuration problems to occur on both UP or SMP systems. The most frequently noted configuration problem is memory configuration on SMPs that have four to eight processors.

For the current SMP models (G30, J30, R30), the number of memory banks is determined by the number and type of memory cards in the system. For example, a 64 MB memory card can contain a single memory bank. If a system with

eight processors was running with a memory card that contained a single memory bank, contention would occur for the memory bank. Therefore, adding memory cards would increase the system performance.

Since SMP and UP systems experience common performance problems, it is good practice to examine the system for obvious problems such as I/O bottlenecks, insufficient memory (paging), and networking problems before assuming that an SMP system problem exists.

Heuristics

Heuristics can help diagnose performance problems since several pathologies are possible on SMP systems. The following guidelines provide some help for choosing tools to study an SMP system that has performance problems.

First, if the CPU is busy, the processor cycles are not idle. First view the user/system mix using `iostat`, `vmstat`, or `sar`. If the problem is excessive user time, try using `tprof` to determine which threads and functions within the threads are consuming the CPU. If the problem is in system time, try trace-based analysis. Collect a trace and use `utld` on the trace. (See "utld—A Trace-based Performance Tool" in this issue.) It can also be helpful to run `lockstat` to view kernel locking behavior.

Second, if the CPU is not busy, a problem exists with I/O balance or event synchronization. First look for non-SMP specific problems, such as disk I/O bottlenecks, network problems, and excessive paging. If this is negative, the problem is most likely related to event synchronization, which requires trace-based analysis.

The rest of this article contains synthesized versions of SMP problems analyzed by the AIX performance teams. All of the measurements given were generated on a 4-way G30 SMP system running AIX 4.1.4.

Case 1: Parallelize FORTRAN Programs

A performance problem was reported during the evaluation of a preprocessor to parallelize FORTRAN programs into threads. Poor scaling was reported on the resulting parallelized program. The first step in analyzing the problem was to view the CPU consumption of the program using `timex -s (sar)` shown in Figure 2.

```
AIX longestday 1 4 00000000A600   12/05/95

12:47:57     %usr    %sys    %wio   %idle
12:48:41     36      43      0      21

12:47:57     bread/s   lread/s   lrcache   bwrit/s   lwrit/s   %wcache   pread/spwrit/s
12:48:41     0         0         0         0         0         0         0

12:47:57     slots    cycle/s  fault/s  odio/s
12:48:41     126445   0.00     5.15     0.00

12:47:57     rawch/s  canch/s  outch/s  rcvin/s  xmtin/s  mdmin/s
12:48:41     0        0        0        0        0        0

12:47:57     scall/s  sread/s  swrit/s  fork/s  exec/s  rchar/s  wchar/s
12:48:41     18734    21       0        0.05    0.07    89850    36

12:47:57     cswch/s
12:48:41     13737

12:47:57     iget/s   lookuppn/s   dirblk/s
12:48:41     4        1            2

12:47:57     runq-sz   %runocc   swpq-sz   %swpocc
12:48:41     3.4       100       1.0       100

12:47:57     proc-sz     inod-sz    file-sz   thrd-sz
12:48:41     41/131072   545/1324   203/255   48/262144

12:47:57     msg/s   sema/s
12:48:41     0.00    0.00
```

Figure 2. Case 1 sar measurements

The results are surprising. First, we have high system utilization in an application with almost no I/O. Second, we have a high context switch rate. It appears that the application has a serialization problem. Threaded applications often serialize with mutex locks. Mutex locks are supported in AIX's `pthread` library and via specialized kernel services.

Further analysis with `tprof` shows that the application has significant time in `libpthreads`, particularly in the functions `_spin_lock`, `pthreads_mutex_lock`, `pthread_mutex_unlock`, and `_spin_unlock`. This confirms that the application is doing serialization via the `pthreads` library.

After viewing an AIX trace using `trcrpt`, (Figure 3) we see threads going to sleep via `thread_waitlock`. In the sequence, the thread running on CPU 3 calls `thread_waitlock` because of lock contention detected in the `pthreads` library. This causes the processor to make the thread sleep (via `e_assert_wait` and `e_block_thread`) and dispatch a new thread (the idle thread).

`Utld` shows the CPU consumption caused by `thread_waitlock` and `thread_unlock`. Figure 4 shows that `thread_waitlock` and `thread_unlock` dominate system call CPU time. It points to 28.967% plus 19.285% as the largest slice of time.

Kernel (System Call) Summary

The program did not scale because the locking was not granular enough. As threads attempted to get the mutex lock, which managed shared data among the threads in the program, they ended up spinning for the lock because it was heavily used. The AIX mutex lock implementation limits the duration of the lock spin. When the limit is hit, the thread is made to sleep. Figure 5 shows how threads contend for the mutex lock over time.

The parallelizing application did not make any efforts to determine the appropriate amount of parallelism, but it required the user to direct it to parallelize the important loops. If the user encouraged the application to parallelize a loop that was not inherently parallel, the locking overhead to manage the shared data in the resultant program was greater than the benefit of the parallelism.

```
ID CPU     ELAPSED_SEC       DELTA_MSEC      APPL            SYSCALL KERNEL   INTERRUPT
101 3      0.000679296       0.002688        thread_waitlock LR = D027BF34
112 3      0.000694528       0.011264        lock:           miss lock
 addr=20000BD8 lock status=30000000 requested_mode=LOCK_SWRITE return addr=36F4
name=0000.0000
46D 3      0.000699904       0.005376        wait_on_lock:   pid=5874 tid=798
1 lockaddr=20000BD8
460 3      0.000705664       0.005760              e_assert_wait:   tid=798
1 anchor=2FF3BCFC flag=1 lr=19084
462 3      0.000738816       0.027776              e_block_thread:   tid=79
81 anchor=2FF3BCFC t_flags=0020 lr=19094
10C 3      0.000848512       0.009472                  dispatch:   idle process
  pid=1290 tid=1291 priority=127 old_tid=7981          old_priority=103 CPUID=3
```

Figure 3. Case 1 trcrpt output

Processing Total (msecs)	Percent Processing Time	Count	Path in msecs min	avg	max	System Call
51873.955	28.967	000004291	0.016	0.437	1.469	thread_waitlock
1247.599	19.285	000003665	0.019	0.340	1.126	thread_unlock
3.340	0.052	000000009	0.046	0.371	1.090	kioctl
1.809	0.028	000000001	1.809	1.809	1.809	_exit

Figure 4. Case 1 utld system call output

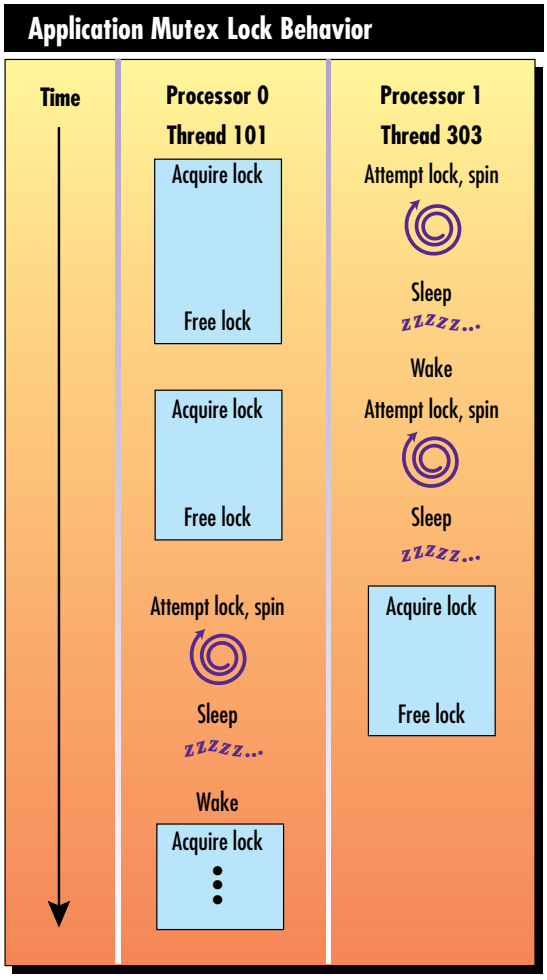


Figure 5. Application mutex lock behavior

Case 2: Pseudo-Database Application

A pseudo-database application did not scale well on a 4-way SMP system. The application had many processes accessing a common database—with high I/O rates. Adding more users did not increase throughput. The first view of the workload using `iostat` (Figure 6) shows excessive time in the AIX kernel. Further viewing via `vmstat` indicates no paging. The I/O rates are not too high, although it appears that application caching of reads is effective because all disk I/Os are writes. Since the CPU is busy, it is possibly a locking problem.

An AIX trace provides a more detailed view of the system. Using `trcrpt` to view the trace allows interesting events for threads and processors to be tracked. Figure 7 shows that even without lock instrumentation turned on, evidence of lock conflict may be noted in the trace. Notice the event `wakeup_lock`, a kernel function used to wake threads waiting on a lock. When the read system call wakes up a thread to run (observable by the `setrq` trace hook), 89 threads are waiting on the lock! At this stage, it would be wise to enable lock instrumentation and use `lockstat`.

Figure 8 shows `lockstat` output on the running system. The `PROC_INT_CLASS` and `IRDWR_LOCK_CLASS` locks stand out. The `PFS IRDWR_LOCK_CLASS` lock indicates a filesystem

```
tty     tin     tout     avg-cpu:  % user  % sys  % idle  % iowait
        0.0     1.6                26.2    47.6   2.9     23.3

Disks:     % tm_act   Kbits/sec  tps       Kb_read   Kb_wrtn
hdisk0     20.1       73.4       17.7      0         369
hdisk1     14.5       74.0       18.5      0         372
```

Figure 6. Case 2 iostat output

```
ID   CPU    ELAPSED_SEC      DELTA_MSEC    APPL  SYSCALL KERNEL   INTERRUPT
101  2      0.007946240      0.247296      lseek LR=100005A0
104  2      0.007965184      0.018944      return from lseek [18 usec]
101  2      0.007975680      0.010496      kreadv LR=D00437C8

163  2      0.007989888      0.014208      read(3,2FF21CB,1000)
10A  2      0.007999232      0.009344      PFS rdwr (vp,ip)=(9AB9880,9AB98B0)
10A  2      0.008531200      0.011776      PFS rdwri VA.S=1D9000.BA17 bcount=1000 ip=9AB98B0
10A  2      0.008689408      0.045312      setrq: cmd=inode pid=37782 tid=38821 priority=65 policy=0
46E  2      0.008711936      0.022528      wakeup_lock: lockaddr=9AB9930 waiters=89
104  2      0.008734208      0.008320      return from kreadv [758 usec]
```

Figure 7. Case 2 trcrpt output

Subsys	Name	Ocn	Ref/s	%Ref	%Block	%Sleep
PROC	PROC_INT_CLASS	1	5128	12.37	10.29	0.00
PFS	IRDWR_LOCK_CLASS	18943	4041	9.75	80.28	9.00

Figure 8. Case 2 lockstat output

problem with file inode locks. UNIX file semantics require a lock to be held over I/Os. If many processes are reading or writing to a single file, contention to accessing the file may result. This is not strictly an SMP problem; it could also occur on a uniprocessor.

The use of `PROC_INT_CLASS` lock is pushed higher in the system because lock contention is occurring on the `IRDWR_LOCK_CLASS`. The `PROC_INT_LOCK` lock is used for dispatches on AIX 4.1.4, so if lock contention causes extra dispatching, extra conflict occurs on the dispatching lock. The solution is to use raw logical volumes instead of files. Otherwise, try to split the database into multiple files to reduce inode lock contention on any single file.

Figure 9 shows how contention for the inode lock slows application performance.

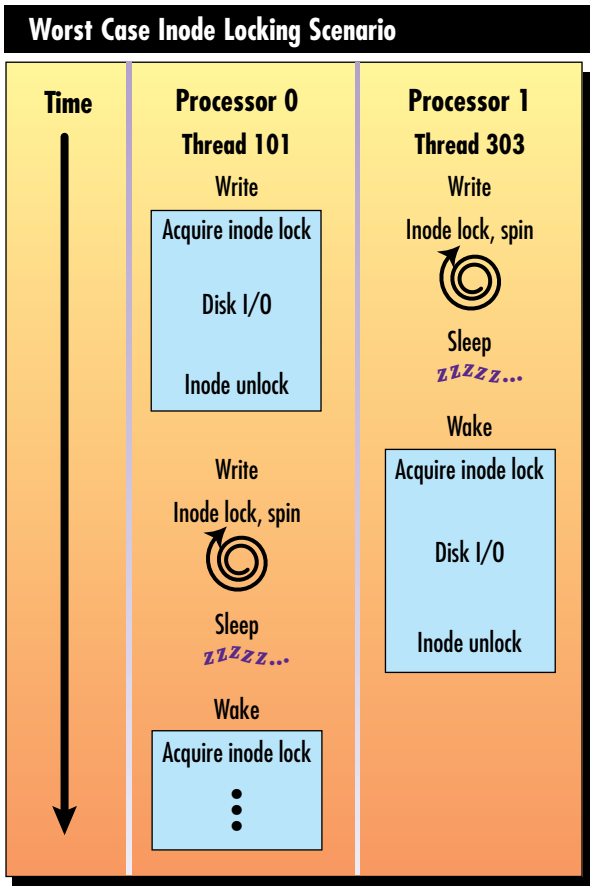


Figure 9. Worst case inode locking scenario

Case 3: Pseudo-database Application

This pseudo-database case also did not scale well to SMP. Figure 10 shows `iostat` measurements on the application. With the high kernel CPU consumption, it appeared that AIX had a problem. `Sar` (not shown) shows a high context switch rate of 13767 context switches per second.

All workloads that show high kernel time provide an ideal opportunity to try `utld` because

```
tty:    tin    tout    avg-cpu:  % user    % sys    % idle    % iowait
        0.0    0.0                19.8      60.5     19.8      0.0

Disks:          % tm_act     Kbits/sec     tps     Kb_read    Kb_wrtn
hdisk0          0.0          0.0           0.0     0          0
hdisk1          0.0          0.0           0.0     0          0
```

Figure 10. Case 3 iostat output

Processing Total (msecs)	Percent Processing Time	Count	Path in msecs min	avg	max	System Call
5424.494	63.181	000004760	0.014	1.140	13.730	semop
7.193	0.084	000000010	0.198	0.719	0.904	kwritev

Figure 11. Case 3 utld system call output

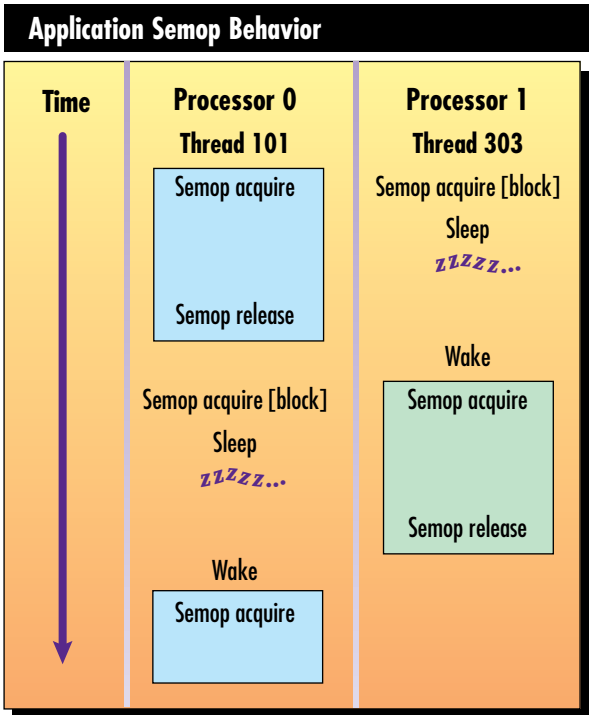


Figure 12. Application semop behavior

the kernel CPU utilization should show up in system calls or interrupts. Figure 11 shows the `utld` output for system calls. The `semop` system call is using 63% of all processing time.

Kernel (System Call) Summary

We discovered during discussions with the application developers that a single semaphore was being used to serialize all access to a shared memory segment. Collisions on the semaphore were limiting throughput. As in Case 1, the granularity of locking via the semaphore needed to be improved in order to increase the scalability of the application. Figure 12 shows how semaphore contention slows application performance.

Case 4: Server/Client Processes

The final case was an application composed of a server process and client processes. The client processes communicate with the server through shared memory. When a client process has work for the server, it sends a message. The client then waits for the server on a unique (to that client) semaphore. Figure 13 shows `sar` data for

```
AIX longestday 1 4 00000000A600    12/05/95

15:34:14   %usr    %sys    %wio    %idle
15:34:15   22      6       0       72

15:34:14   bread/s   lread/s   %rcache   bwrit/s   lwrit/s   %wcache   pread/s   pwrit/s
15:34:15   0         0         0         0         0         0         0         0

15:34:14   slots    cycle/s    fault/s    odio/s
15:34:15   126410   0.00       181.78     0.93

15:34:14   rawch/s   canch/s   outch/s   rcvin/s   xmtin/s   mdmin/s
15:34:15   0         0         0         0         0         0

15:34:14   scall/s   sread/s   swrit/s   fork/s   exec/s   rchar/s   wchar/s
15:34:15   15809     116       41        6.85     0.31     418546    828

15:34:14   cswch/s
15:34:15   3019

15:34:14   iget/s   lookuppn/s   dirblk/s
15:34:15   27       11           14

15:34:14   runq-sz   %runocc   swpq-sz   %swpocc
15:34:15   4.7       100       1.0       100

15:34:14   proc-sz      inod-sz      file-sz      thrd-sz
15:34:15   41/131072    621/1520     204/340      48/262144

15:34:14   msg/s      sema/s
15:34:15   5102.93    5116.01
```

Figure 13. Case 4 sar output

Processing Total (msecs)			Percent of Total Processing Time			Process Name
Combined	Application	Kernel	Combined	Application	Kernel	(process id/thread id)
2251.938	1630.816	621.122	23.808	17.242	6.567	lazyp (5874 5683)
44.737	0.086	44.650	0.473	0.001	0.472	syncd (3044 2003)
18.345	2.350	15.995	0.194	0.025	0.169	lazyp (5876 5685)
17.973	2.472	15.501	0.190	0.026	0.164	lazyp (17610 7435)

Figure 14. Case 4 utld output

the workload. Note that a significant amount of idle time occurs in the workload.

At this stage, the CPU system utilization seems limited to some integral fraction of the number of processes in the system. This could imply a single-server bottleneck characteristic of workload imbalance. The system has four processors with 28% CPU utilization. Figure 14 shows that the trace and `utld` indicate one process, `lazyp`, is consuming 23.8% of the system's CPU.

The server process does most of the work in this workload. Since it can only run on one processor at a time, the other processors are underutilized. To achieve higher throughput on an SMP system, the server process must be split into independent processes or threads.

Figure 15 shows how a single server bottleneck limits performance for an application.

Conclusion

Three success factors are important to effectively study the performance of an SMP system. First, understand the capabilities of the performance tools. This is important because a vast array of choices are available, but each presents a different view of the system. Second, use heuristics to select the tools wisely. Most SMP problems are not subtle, so with good insight you should be able to understand them quickly. Third, network with others. It is very likely that someone else has seen the identical or similar problem in the past.

Bret R. Olszewski, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Olszewski is a senior programmer working on MP performance. He joined IBM in 1989 and has worked on various aspects of AIX performance. He has a BS in Computer Science from the University of Minnesota.

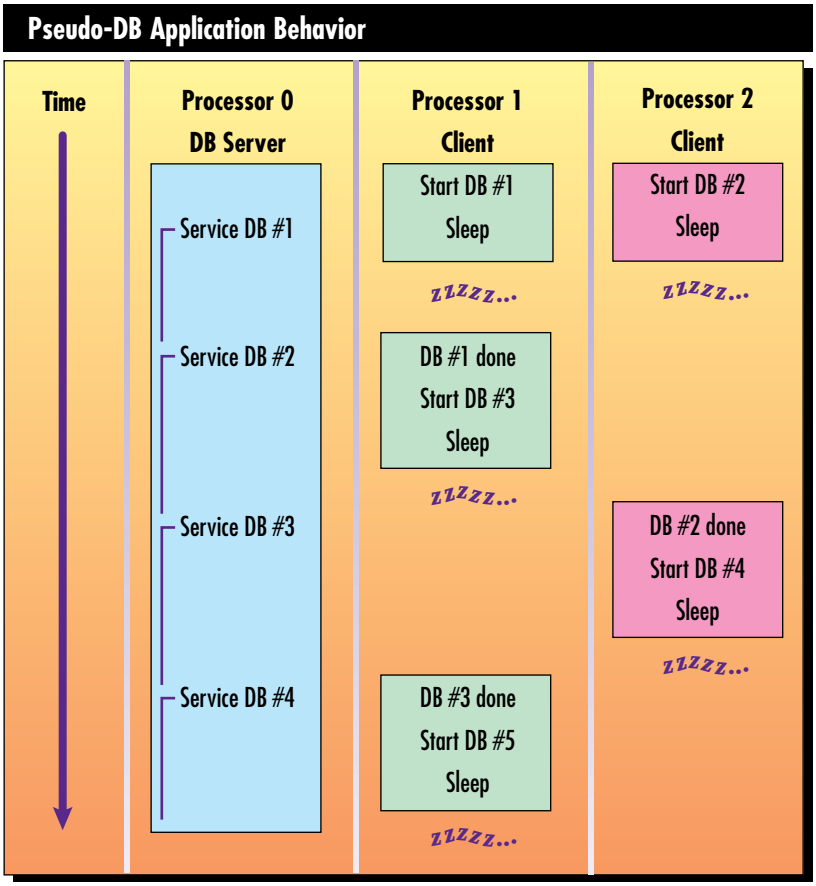


Figure 15. Single-server bottleneck limits application performance

Jim Van Fleet, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Van Fleet is a senior programmer in AIX Performance in Austin. He has held several technical and managerial positions during his career at IBM including broad experience in operating systems with a specialty in Symmetric Multiprocessing systems. Mr. Van Fleet has a BS in Mathematics from Michigan State University and an MS in Computer Science from Union College.