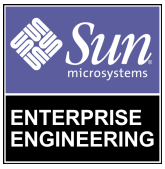
Solaris Resource Manager™ - Decay Factors and Parameters

By Richard McDougall - Enterprise Engineering

Sun BluePrints™ OnLine - April 1999



http://www.sun.com/blueprints

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300 fax 650 969-9131

Part No.: 806-3840-10
Revision 01, April 1999

Please
Recycle

Solaris Resource Manager™—Decay Factors and Parameters

It is sometimes important to understand how the Solaris Resource Manager™ product allocates resources to processes, to help explain some of the factors that affect the amount of resources allocated to different processes. This article explores how the Solaris Resource Manager scheduler works.

The Solaris Resource Manager Scheduler

Solaris Resource Manager product is designed to run with minimal or no tuning. But there are a number of configurable parameters that govern the way the scheduler apportions shares. These parameters can be used to influence the way the scheduler reacts to dynamic workloads, and different parameters can sometimes benefit particular workloads.

The Solaris Resource Manager product scheduler is a complete replacement for the Solaris™ Operating Environment timeshare scheduler (TS), and hence does not use any of the existing parameters provided with the TS scheduler module. On systems where Solaris Resource Manager is installed, use the `dispadmin` command to display the configured scheduling classes:

```
# dispadmin -l
CONFIGURED CLASSES
==================

SYS     (System Class)
SHR     ((SHR) SRM Scheduler)
TS      (Time Sharing)
IA      (Interactive)
```

At boot time, the Solaris Operating Environment kernel starts all non-kernel processes in the SHR class instead of the traditional TS class. You can see this with the ps command.

```
# ps -cafe
     UID   PID  PPID  CLS PRI    STIME TTY      TIME CMD
    root     0     0  SYS  96   Mar 12 ?        0:00 sched
    root     0     0  SYS  96   Mar 12 ?        0:00 sched
    root     1     0  SHR  59   Mar 12 ?        1:03 /etc/init -
    root     2     0  SYS  98   Mar 12 ?        0:00 pageout
    root     3     0  SYS  60   Mar 12 ?        7:19 fsflush
    root     4     0  SYS  60   Mar 12 ?        1:05 srmgr
   root  563    1  SHR  60   Mar 12 console  0:00 /usr/lib/saf/ttym
    root   313     1  SHR  59   Mar 12 ?        0:04 /usr/lib/utmpd
   root   201    1  SHR  59   Mar 12 ?       0:00 /usr/sbin/keyserv
   root    14    1  SHR  59   Mar 12 ?       0:00 vxconfigd -m boot
    root   159     1  SHR  59   Mar 12 ?        0:23 limdaemon
   root   199    1  SHR  59   Mar 12 ?       0:00 /usr/sbin/rpcbind
    root   252     1  SHR  59   Mar 12 ?        1:24 automountd
ctd...
```

The TS scheduler is configured by altering the dispatcher table that controls how process priorities are decayed. The TS scheduler uses a variable time quantum and adjusts process priorities as they expire their time quantum. You can observe the existing dispatcher table with the dispadmin command.

```
# Time Sharing Dispatcher Configuration
RES=1000

# ts_quantum  ts_tqexp  ts_slpret  ts_maxwait ts_lwait  PRIOR LEVEL
       200         0        50           0        50        #     0
       200         0        50           0        50        #     1
       200         0        50           0        50        #     2
       200         0        50           0        50        #     3
       200         0        50           0        50        #     4
(snip)
        40        44        58           0        59        #    54
        40        45        58           0        59        #    55
        40        46        58           0        59        #    56
        40        47        58           0        59        #    57
        40        48        58           0        59        #    58
        20        49        59       32000        59        #    59
```

Process priorities are shown by the priority levels, which range from 0 through 59 The highest priority process (represented as 59) gets the most amount of CPU. A process starts with a normal priority inherited from its parent somewhere in the middle of the table. If the process uses its time quantum (each priority level has a different quantum represented by ts_quantum), it will be decayed to the priority in the ts_tqexp column. A process that has had no CPU will have its priority raised to ts_lwait, and a process that has awakened after sleeping for an I/O will have its priority set to ts_slpret.

Decay Algorithms

If you were to use a fixed priority scheduler with no decaying of process priorities, a higher priority process would always get as much CPU as it wanted and could block all other lower priority processes from receiving any CPU. (The Solaris Operating Environment RT scheduler works this way.) The decay algorithm prevents any one process from hogging all of the CPU by decrementing its priority once it receives its allocation of CPU. This provides a fair allocation of CPU resources to each process, based on its priority. The amount of CPU allocated to each process can be influenced by setting the `nice` value between -19 and +19. The `nice` value is added to the process priority as it is scheduled. It allows one process to receive more CPU than another, but because of the decay model, it will not completely block other processes.

The table-based decay algorithms used in the Solaris Operating Environment software provide a flexible framework where the scheduler can be tuned for different workloads. The scheduler also replaces the BSD style scheduler's notion of a simple decay that does not use a table to calculate process priorities.

Comparison to BSD Style Priority Decay

The BSD UNIX® scheduler decays the priority of each process by multiplying its priority by a fixed decay factor at a regular interval:

*process_pri = 0.66 * process_pri + nice_factor*

We can compare what would happen if we varied the decay factor. At one end of the range, we could set the decay factor to 0.9 (slow decay), which would mean we almost have fixed process priorities. This would cause a process with a high priority to stay at a high priority for a long time and could allow a process to monopolize the CPU and starve other processes. At the other end of the range, we could decay process priorities very fast by using a decay factor of 0.1. This would prevent CPU monopolization but would cause all processes to receive similar amounts of CPU regardless of their nice factor. It would also allow a user who runs a large number of processes to get a large and unfair amount of CPU.

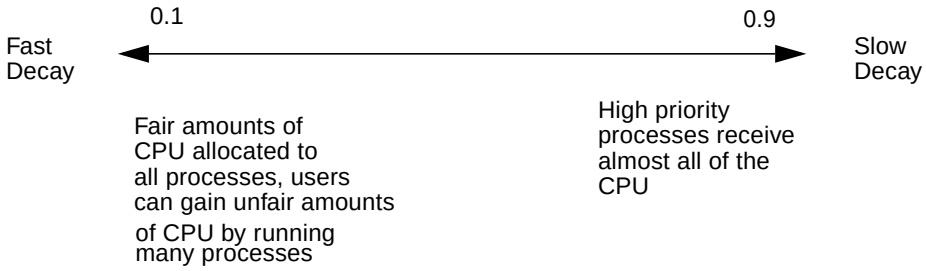


FIGURE 1 The Effect of Decay Time on Allocation Fairness

Usage Decay and Process Priority Decay

The Solaris Resource Manager scheduler has two different notions of decay that affect processes: one is the usage decay and the other is the process priority decay. This happens because Solaris Resource Manager software is implemented with two schedulers, the user scheduler and the process scheduler, each with its own set of parameters. The user scheduler calculates and attempts to bias *users* so that they each receive their share of allocated CPU according to the amount of shares allocated to them. The process scheduler sits underneath the user scheduler and manages fine-grained allocation of CPU to each process by using a BSD style decay model.

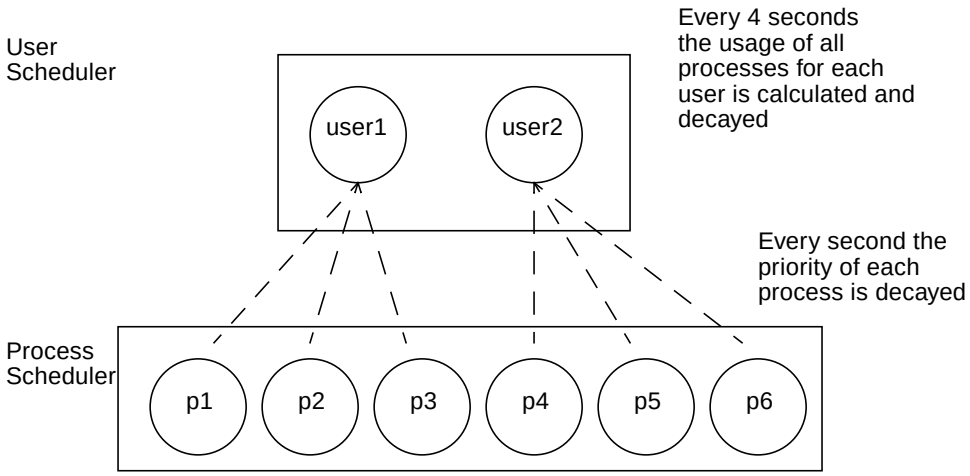


FIGURE 2 Two Levels of Scheduling in Solaris Resource Manager

Process Priority Decay

If we ignore the user scheduler for the moment, we can look at the process scheduler as just another UNIX process scheduler that uses decay factors. The SHR scheduler does not use table driven decay. It uses a model closer to the BSD decay where each processes priority is decayed according to a fixed decay factor at regular intervals (each second). Internal process priorities in the Solaris Resource Manager process scheduler are known as userpri and range from 0 to 10^{32}, where 0 is the highest priority

FIGURE 3 The Effect of Decay on Process Priority

To prevent CPU monopolization, a low priority process (a high userpri value) has its userpri decayed each second by the fixed decay parameter so that it will eventually get some allocation of CPU. This prevents a process from CPU starvation in a similar way to the BSD scheduler. The process priority decay parameter can be seen with the srmadm command:

```
# srmadm show -V3
Scheduling flags = -share, -limits, -adjgroups, -limshare, -fileopen
Idle lnode = root
Lost lnode = NONEXISTENT

Usage decay rate half-life = 120.0 seconds,
       (0.977159980000 - 4 second units, 0.999942239403 - 0 Hz units),

max. users                  = 12
active users                = 0
active groups               = 0
scheduler run rate          = 4 seconds
number of configured lnodes = 0

Process priority decay rate biased by "nice":-
   high priority (nice -20) 0.4044 (half-life   0.8 seconds)
average priority (nice   0) 0.7039 (half-life   2.0 seconds)
    low priority (nice  19) 0.9885 (half-life  60.0 seconds)

Minimum group share factor         = 0.750
Maximum user share factor          = 2.000
Async-nice share factor            = 1.000e+07

Max. value for normal usage        = 1.000e+12
Max. value for normal p_sharepri   = 1.000e+28
Max. value for idle p_sharepri     = 1.000e+34

Cache hits on valid lnodes         = 0
Cache hits on invalid lnodes       = 0
Cache misses                       = 0
Number of lnodes replaced          = 0

Default system flags =  -admin, -uselimadm, -onelogin, -nologin,
                        -asynckill, -asyncnice, -noattach
```

The decay parameter for normal processes is 0.7039 by default. This means that every two seconds the userpri value for the process will halve (referred to as the half-life). The `nice` command can be used to alter process priorities. The decay values for processes depend on the `nice` factor assigned to each process. To give a process a higher priority, you can use the nice command to influence the decay value. For example running a command with `nice -19` will cause the processes

sharepri to drop faster. This is because you now multiply by a factor of 0.404, which will allow the process to wait a shorter time before it gets access to the CPU again

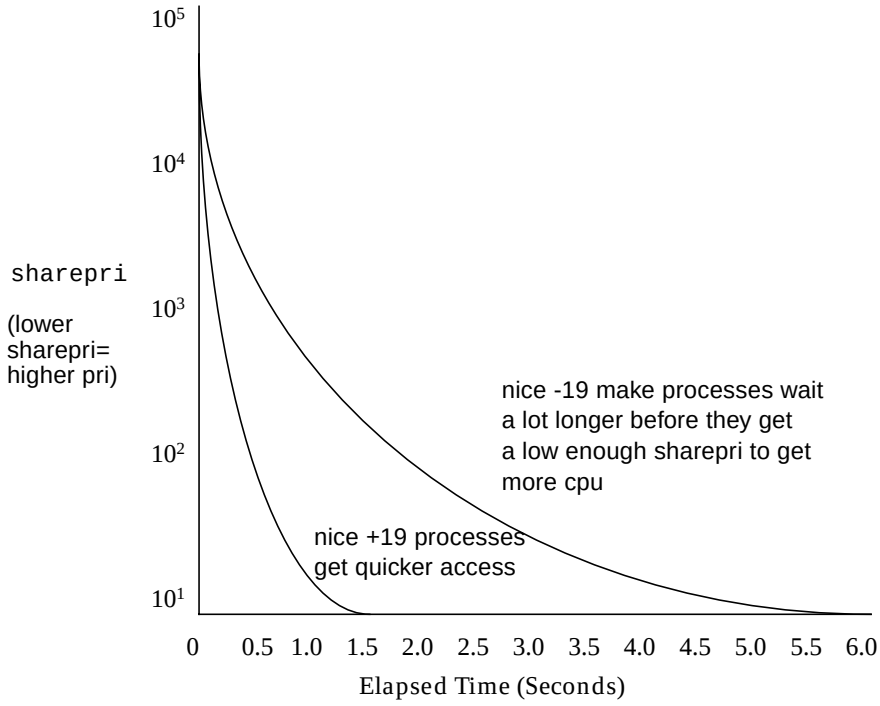


FIGURE 4 Process Priority Decay with Two Different nice Values

You can see how a process's sharepri is decayed every second using the priority decay factor. Now let's look at how process sharepri is increased. The sharepri for each process is increased at the same time according to the amount of CPU that it has used, just as we saw with the BSD scheduler equation. But the SHR scheduler includes a user usage factor that is derived from the user scheduler shown in FIGURE 2. This is how SHR implements the notion of user based scheduling rather than just process-based scheduling. At each second, the scheduler manipulates the sharepri of each process by the following:

*sharepri = sharepri * decay_factor + users_usage_factor*

Consider a process that is using small amounts of CPU over the same period shown in FIGURE 4. You can see that the sharepri for each process increments as the process uses CPU and then decays while it does not.

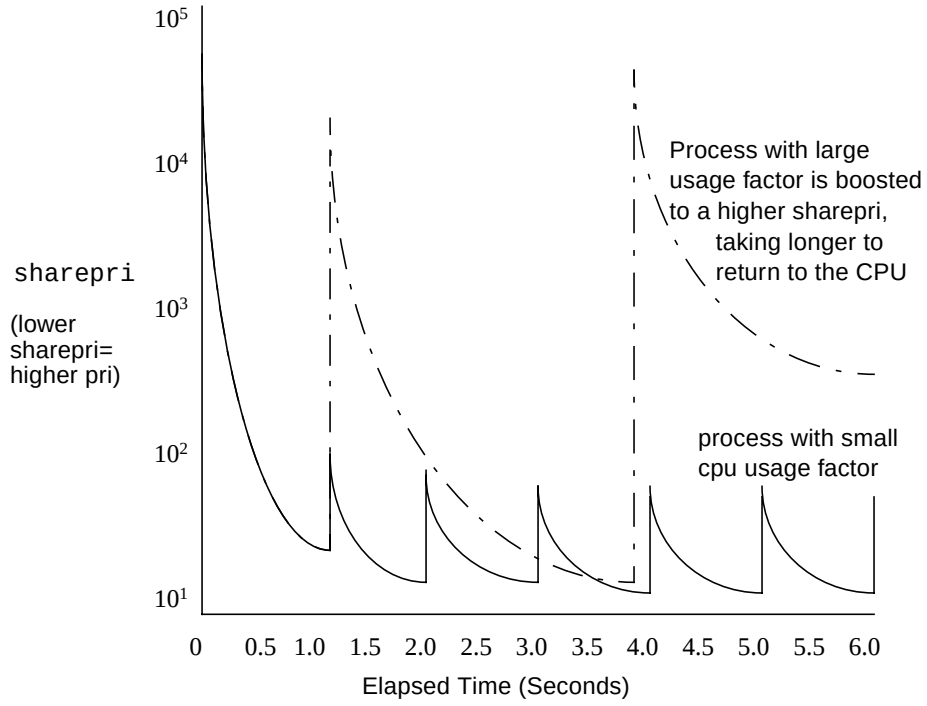


FIGURE 5 Process Priority Increase and Decay

A Note about `/bin/ksh`

The korn shell automatically sets the nice value of processes when they are sent to the background. This will result in a slightly different process decay rate for foreground and background processes.

```
  PID  USERNAME THR PRI NICE  SIZE   RES STATE   TIME    CPU COMMAND
 1599 user1      1  59    0  896K  560K cpu/0   0:31 80.44% t1spin
  425 user2      1  15    4  896K  560K run     0:28  1.50% spin
```

Usage Decay

So far we have focused on the process scheduler (bottom layer of FIGURE 2) and for the purpose of simplification, we viewed the process scheduler as just another UNIX scheduler. The fair share features of Solaris Resource Manager scheduler are implemented by factoring in the user usage, which is calculated in the top user

scheduler layer. The user scheduler is the most important and visible portion of Solaris Resource Manager and implements usage decays which control long term CPU allocation responsiveness. Short term (less then 4 second) responsiveness and individual process importance is controlled by the process priority decay parameters discussed in the previous section.

The user scheduler summarizes process usage across users and compares it to the shares allocated with the Solaris Resource Manager policy hierarchy. This information is summarized as normalized usage that is passed to the process scheduler as a usage factor to control all of the processes within each user. Processes within a user may be prioritized individually using the nice command to influence their priority decay.

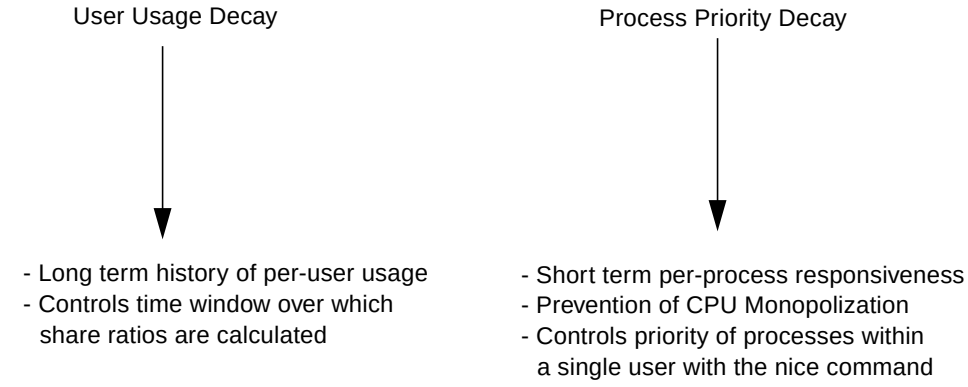


FIGURE 6 Comparison of Usage Decay and Priority Decay

The user scheduler runs every four seconds and calculates the total CPU usage for all of the processes for each user. The usage is added to the usage and accrued fields in the lnode for that user. You can see the usage increment as a process executes by looking at the usage and accrued fields in the lnode with the liminfo command.

```
# liminfo -c chuck
Login name:                chuck      Uid (Real,Eff):        2001 (-,-)
Sgroup (uid):           root (0)      Gid (Real,Eff):         200 (-,-)

Shares:                       50      Myshares:                     1
Share:                        41 %    E-share:                      0 %
Usage:                    142302       Accrued usage:          4.23e+10
```

An example is a process that has been using CPU continuously for two seconds. This would incur a usage increment of 2 seconds x 1000 (Solaris Resource Manager charge per tick) x 100 (ticks per second) = 20000.

If you were to allocate CPU resources based on instantaneous usage, you would only be able to allocate the proper shares to each user if both users were using CPU at the same time. In reality, this rarely happens unless both controlled users are running continuous batch type jobs. More often, users use small portions of CPU at different intervals, and you want to allocate shares of CPU based on usage over a broader period of time. For example, if user 1 is using one second of CPU every ten seconds and user 2 is using CPU continuously, you would get incorrect results if you controlled the instantaneous CPU allocated to each user.

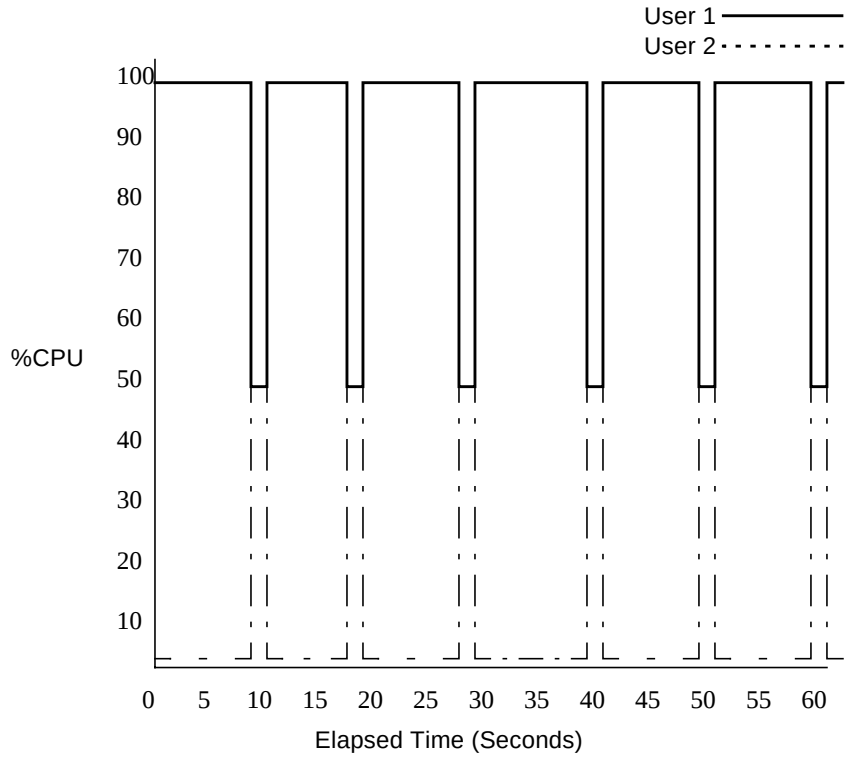


FIGURE 7 CPU Allocation with Controlled Instantaneous Use

You can see in FIGURE 7 that if we allocate 50 shares to user 1 and 50 shares to user 2 described previously and control their CPU instantaneously user 1 would get 100 percent of the CPU while it is executing on its own, and then each user would get 50 percent while they are both busy. This means that even though user 2 is only using 10 percent of the CPU, it is still being constrained by the resource manager. And user 2's one-second request every 10 seconds ends up taking two seconds.

By adding a decay factor, you can average the CPU use over a larger time window. This gives a true view of each user's average use. If we revisit our example, we could set a decay factor so that usage history is collected over a larger time window,

say 30 seconds. This would mean that user 1 would be recorded as using 90 percent of the CPU and user 2 would be recorded as using 10 percent. User 2 could use almost 100 percent of the CPU when it requests, and the one-second request would complete in one second

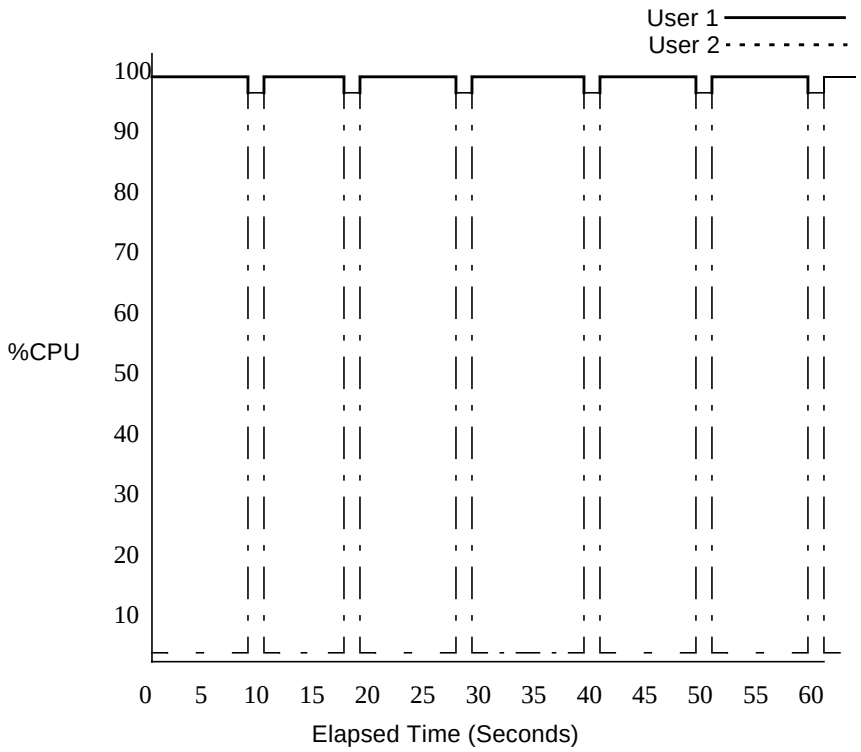


FIGURE 8 CPU Allocation with Decay to Include Usage History

User 1 can use close to 100 percent rather than 90 percent because of the process priority decay and the responsiveness of the process scheduler. Making the priority decay longer allows short term bursty users to get access to the CPU when they need it. This is particularly useful for interactive users or web servers.

If you were to keep too much usage history, then a user could be completely blocked from CPU resources for a long period if they over-consumed their share. This would happen if we set a very long decay time (almost no decay) and a user used more than its share while nobody else was using their share. For example, if user 1 used 100 percent of the CPU over a weekend when no other users were on the system, when other users came back to work on Monday morning user 1 could stop for up to two whole days. Because of this, you need to choose a decay time that allows the scheduler to get the most useful average usage but is not too long to cause marooning of users.

```
# srmadm show -V3
Scheduling flags = -share, -limits, -adjgroups, -limshare, -fileopen
Idle lnode = root
Lost lnode = NONEXISTENT

Usage decay rate half-life = 120.0 seconds,
       (0.977159980000 - 4 second units, 0.999942239403 - 0 Hz units),

max. users                  = 12
active users                = 0
active groups               = 0
scheduler run rate          = 4 seconds
number of configured lnodes = 0

Process priority decay rate biased by "nice":-
   high priority (nice -20) 0.4044 (half-life   0.8 seconds)
average priority (nice   0) 0.7039 (half-life   2.0 seconds)
    low priority (nice  19) 0.9885 (half-life  60.0 seconds)

Minimum group share factor         = 0.750
Maximum user share factor          = 2.000
Async-nice share factor            = 1.000e+07

Max. value for normal usage        = 1.000e+12
Max. value for normal p_sharepri   = 1.000e+28
Max. value for idle p_sharepri     = 1.000e+34

Cache hits on valid lnodes         = 0
Cache hits on invalid lnodes       = 0
Cache misses                       = 0
Number of lnodes replaced          = 0

Default system flags =  -admin, -uselimadm, -onelogin, -nologin,
                        -asynckill, -asyncnice, -noattach
```

The usage decay is set on a global basis and can be seen with the srmadm command.

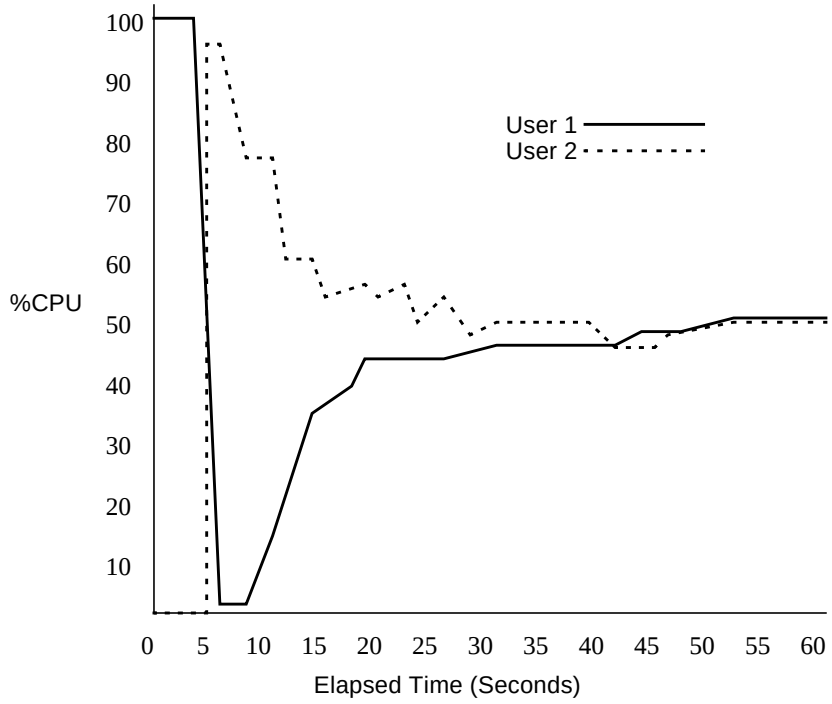


FIGURE 9 Thoughput of a User Starting Affected by Decay

You can see a demonstration of usage history in FIGURE 9. It shows two users with equal shares. User 1 has been consuming CPU for a very long time. A second user, user 2 has not. The second user, user 2, starts using CPU 5 seconds into the measured period. The large CPU usage history of user 1 causes their usage factor to be high (calculated by the user scheduler). When this is passed to the process scheduler, it causes the process to have a high sharepri. Before user 2 starts, user 1 is the only user on the system, and even though user 1 has a high sharepri (low priority) user 1 can consume all of the CPU. When user 2 starts, its usage history is zero, and its usage factor is zero. When this is passed to the process scheduler, it results in a low sharepri (high priority). This means that user 2 can consume most of the CPU for a very short period. As the decay factor decays user 1's large usage history, user 1 is able to get more of the CPU. The default usage decay is 120 seconds, which, as shown in FIGURE 9 causes both users to end up at their equal allocation of shares after a commensurate amount of time.

Note that there are two effects interplaying here. The usage decay allows user 2 to almost monopolize the processor when user 1 over-consumes its share. But the process priority decay decays priorities fast enough so that user 1 still gets a little of the CPU. The process priority (sharepri) is decayed fast enough so that user 1 gets some CPU, but then user 1's sharepri is boosted high because it has a large usage

history. FIGURE 5 shows the interaction of the two processes described in this example. If we wanted to make user 2 consume 100% of the CPU and stop user 1 completely, then we could set a long process priority decay time.

Summary

We have looked at how the internal scheduler of Solaris Resource Manager product operates. This should explain why Solaris Resource Manager product behaves differently with different processes on the system. We saw that there are two decay factors, one for processes and one for users. We also saw how the user decay affects long term user usage allocation and the process decay affects the short term allocation to each process.

Author's Bio: Richard Mc Dougall

Richard has over 11 years of UNIX experience including application design, kernel development and performance analysis, and specializes in operating system tools and architecture.