Chapter 3 Memory Management

Level 1 Page Management

3.4.3 Page-Out Algorithm

- Introduction to page-out algorithm

Solaris internals P.180

The page-out algorithm is controlled by several parameters, some of which are calculated at system startup by the amount of memory in the system, and some of which are calculated dynamically based on memory allocation and paging activity.

The parameters that control the clock hands do two things: they control the rate at which the scanner scans through pages, and they control the time (or distance) between the front hand and the back hand. The distance between the back hand and the front hand is `handspreadpages` and is expressed in units of pages. The maximum distance between the front hand and back hand defaults to half of memory and is capped at 8,192 pages, or 64 Mbytes. Systems with 128 Mbytes or more of memory always default this distance to 8,192 pages, or 64 Mbytes.

- Summary of Page Scanner Parameters

Solaris Kernel-Performance, Observability & Debugging P.63

The table describes the parameters that control the page-out process in the current Solaris and patch releases.

Parameter	Description	Min	Default (Solaris 8)
lotsfree	starts stealing anonymous memory pages	512K	1/64 th of memory
desfree	scanner is started at 100 times/second	minfree	½ of lotsfee
minfree	start scanning every time a new page is created		½ of desfree
throttlefree	page_create routine makes the caller wait until free pages are available		minfree
fastscan	scan rate (pages per second) when free memory = minfree	slowscan	minimum of 64MB/s or ½ memory size
slowscan	scan rate (pages per second) when free memory = lotsfree		100
maxpgio	max number of pages per second that the swap device can handle	~60	60 or 90 pages per spindle
hand-spreadpages	number of pages between the front hand (clearing) and back hand (checking)	1	fastscan
min_percent_cpu	CPU usage when free memory is at lotsfree	4% (~1 clock tick)	of a single CPU

- Scan Rate Parameters (Assuming No Priority Paging)

Solaris internals PP.180-182

The scanner starts scanning when free memory is lower than `lotsfree` number of pages free

plus a small buffer factor, `deficit`. The scanner starts scanning at a rate of `slowscan` pages per second at this point and gets faster as the amount of free memory approaches zero. The system parameter `lotsfree` is calculated at startup as 1/64th of memory, and the parameter `deficit` is either zero or a small number of pages—set by the page allocator at times of large memory allocation to let the scanner free a few more pages above `lotsfree` in anticipation of more memory requests.

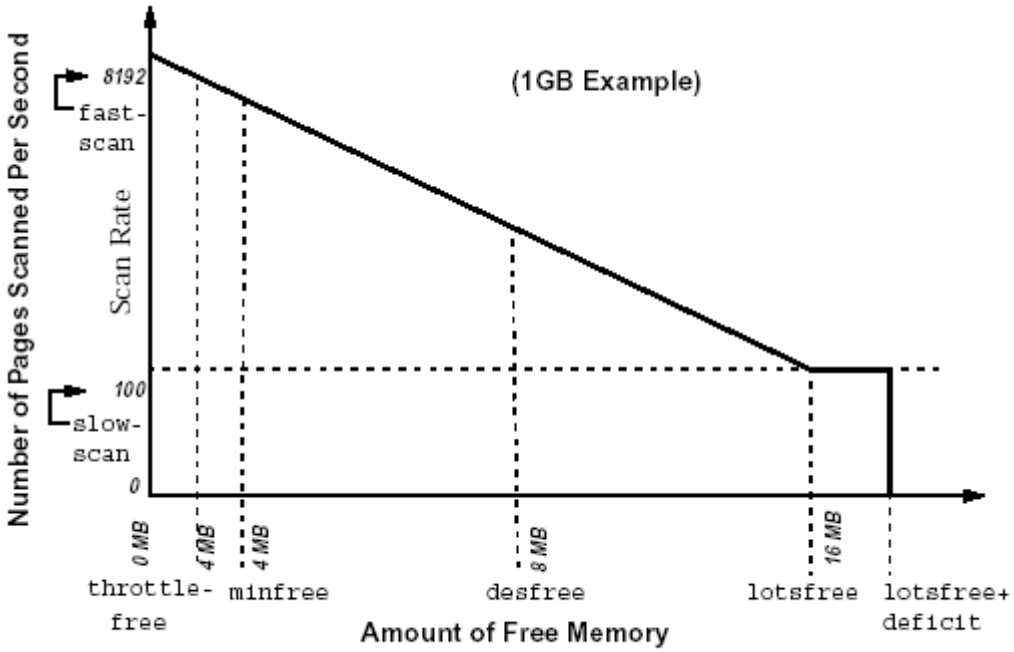

Figure 5.23 *Page Scanner Rate, Interpolated by Number of Free Pages*

Figure 5.23 shows the rate at which the scanner scans increases linearly as free memory ranges between `lotsfree` and zero. The scanner starts scanning at the minimum rate set by `slowscan` when memory falls below `lotsfree` and then increases to `fastscan` if memory falls low enough.

The number of pages scanned increases from the slowest rate (set by `slowscan` when `lotsfree` pages are free) to a maximum determined by the system parameter `fastscan`. Free memory never actually reaches zero, but for simplicity the algorithm calculates the maximum interpolated rate against the free memory ranging between `lotsfree` and zero. In our example system with 1 Gbyte of physical memory (shown in Figure 5.24 on page 185), we can see that the scanner starts scanning when free memory falls to 16 Mbytes plus the short-term memory deficit.

For this example, we'll assume that the deficit is zero. When free memory falls to 16 Mbytes, the scanner will wake up and start examining 100 pages per second, according to the system parameter `slowscan`. The `slowscan` parameter is 100 by default on Solaris systems, and `fastscan` is set to total *physicalmemory*/2, capped at 8192 pages per second. If free memory falls to 12 Mbytes (1536 pages), the scanner scans at a higher rate, according to the page scanner interpolation shown in the following equation:

$$scanrate = \left(\frac{lotsfree - freememory}{lotsfree} \times fastscan\right) + \left(slowscan \times \frac{freemem}{lotsfree}\right)$$

If we convert free memory and `lotsfree` to numbers of pages (free memory of 12 Mbytes is 1536 pages, and `lotsfree` is set to 16 Mbytes, or 2048 pages), then we scan at 2123 pages per second.

$$scanrate = \left(\frac{2048 - 1536}{2048} \times 8192\right) + \left(100 \times \frac{1536}{2048}\right) = 2123$$

By default, the scanner is run four times per second when there is a memory shortage. If the amount of free memory falls below the system parameter `desfree`, the scanner is run at every clock cycle or, by default, 100 times a second. This scheme helps the scanner try to keep at least `desfree` pages on the free list.

- Not Recently Used Time

Solaris internals P.182

The time between the front hand and back hand varies according to the number of pages between the front hand and back hand and the rate at which the scanner is scanning. The time between the front hand clearing the reference bit and the back hand checking the reference bit is a significant factor that affects the behavior of the scanner because it controls the amount of time that a page can be left alone before it is potentially stolen by the page scanner. A short time between the reference bit being cleared and checked means that all but the most active pages remain intact; a long time means that only the largely unused pages are stolen. The ideal behavior is the latter because we want only the least recently used pages stolen, which means we want a long time between the front and back hands.

The time between clearing and checking of the reference bit can vary from just a few seconds to several hours, depending on the scan rate. The scan rate on today's busy systems can often grow to several thousand, which means that a very small time exists between the front hand and back hand. For example, a system with a scan rate of 2,000 pages per second and the default hand spread of 8,192 pages has a clear/check time of only 4 seconds. High scan rates are quite normal on systems because of the memory pressure induced by the file system. (We discuss this topic further in "Is All That Paging Bad for My System?" on page 608.)

- Shared Library Optimizations

Solaris internals P.183

A subtle optimization added to the page scanner prevents it from stealing pages from extensively shared libraries. The page scanner looks at the share reference count for each page; if the page is shared more than a certain amount, then it is skipped during the page scan operation. An internal parameter, `po_share`, sets the threshold for the amount of shares a page can have before it is

skipped. If the page has more than `po_share` mappings (i.e., it's shared by more than `po_share` processes), then it is skipped. By default, `po_share` starts at 8; each time around, it is decremented unless the scan around the clock does not find any page to free, in which case `po_share` is incremented. The `po_share` parameter can float between 8 and 134217728.

- The Priority Paging Algorithm

  Solaris internals PP.183-185

Solaris 7 shipped with a new optional paging algorithm—a page-stealing algorithm that results in much better system response and throughput for systems making use of the file systems. The algorithm is also available on older Solaris releases (from 2.5.1 onwards) with the addition of a kernel patch. You enable the new algorithm by setting the `priority_paging` variable to 1 in `/etc/system`.

```
*
* Enable the Priority Paging Algorithm
*
set priority_paging = 1
```

The new algorithm was introduced to overcome adverse behavior that results from the memory pressure caused by the file system. Back in SunOS 4.0, when the virtual memory system was rewritten, the file system cache and virtual memory system were integrated to allow the entire memory system to be used as a file system cache; that is, the file system uses pages of memory from the free memory pool, just as do processes when they request memory.

The demand paging algorithm allows the file system cache to grow and shrink dynamically as required, by stealing pages which have not been recently used by other subsystems. However, back when this work was done, the memory pressure from the file system was relatively low, as were the memory requests from the processes on the system. Both were in the order of tens to hundreds of pages per second, so memory allocation could be based on who was using the pages the most. When processes accessed their memory pages frequently, the scanner was biased to steal from the file system, and the file system cache would shrink accordingly.

Today, systems are capable of sustaining much higher I/O rates, which means that the file system can put enormous memory pressure on the memory system—so much so that the amount of memory pressure from the file system can completely destroy application performance by causing the page scanner to steal many process pages.

We like to think of the early SunOS 4.0 case as being like a finely balanced set of scales, where the process and file system requests sit on each side of the scale and are reasonably balanced. But on today's system, the scales are completely weighted on the file system side because of the enormous paging rates required to do I/O through the file system. For example, even a small system can do 20 megabytes of I/O per second, which causes the file system to use 2,560 pages per second. To keep up with this request, the scanner must scan at least at this rate, usually higher

because the scanner will not steal every page it finds. This typically means a scan rate of 3,000 or higher, just to do some regular I/O.

As we saw earlier, when we scan at this rate, we have as little as a few seconds between the time we clear and the time we check for activity. As a result, we steal process memory that hasn't been used in the last few seconds. The noticeable effect is that everything appears to grind to a halt when we start using the file system for significant I/O and free memory falls below `lotsfree`. It is important to note that this effect can result even with ample memory in the system—adding more memory doesn't make the situation any better.

To overcome this effect, the page scanner has a new algorithm that puts a higher priority on a process's pages, namely, its heap, stack, shared libraries, and executables. The algorithm permits the scanner to pick file system cache pages only when ample memory is available and hence only steal application pages when there is a true memory shortage.

The new algorithm introduces a new paging parameter, `cachefree`. When the amount of free memory lies between `cachefree` and `lotsfree`, the page scanner steals only file system cache pages. The scanner also now wakes up when memory falls below `cachefree` rather than below `lotsfree`, and the scan rate algorithm is changed accordingly.

$$scanrate = \left(\frac{cachefree - freememory}{cachefree} \times fastscan\right) + \left(slowscan \times \frac{freemem}{cachefree}\right)$$

The scan rate is now interpolated between `cachefree` and zero, rather than between `lotsfree` and zero, as shown in Figure 5.24.

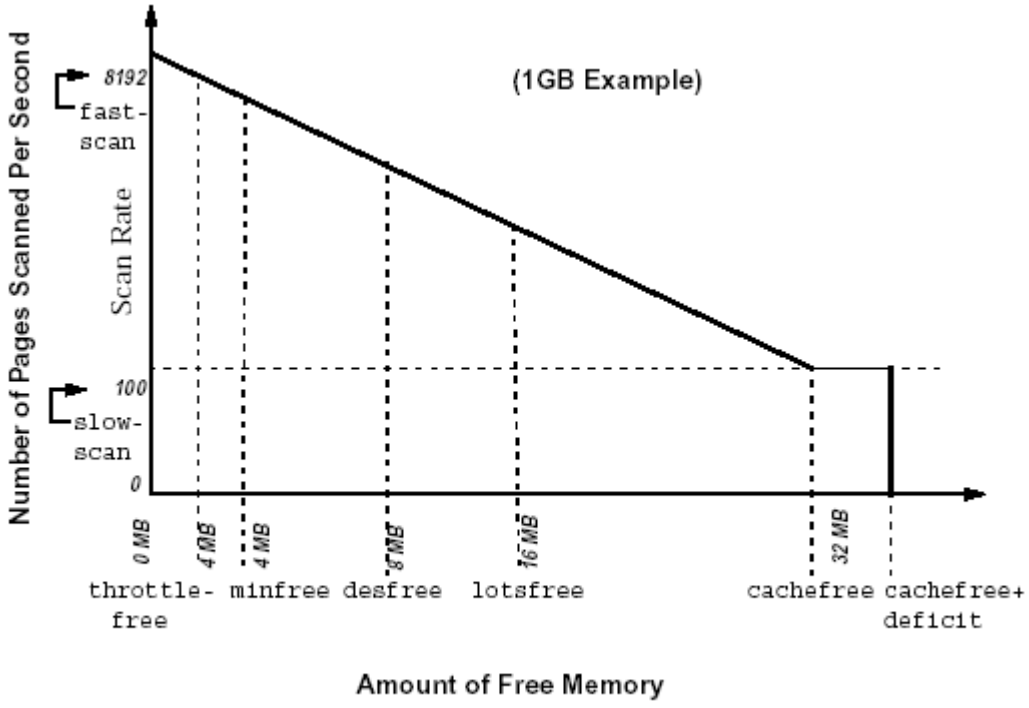

Figure 5.24 *Scan Rate Interpolation with the Priority Paging Algorithm*

The algorithm pages only against the file system cache when memory is between `cachefree` and `lotsfree` by skipping pages that are associated with the swap device (heap, stack, copy-on-write pages) and by skipping file pages that are mapped into an address space with execute permission (binaries, shared libraries).

The new algorithm has no side effects and should always be enabled on Solaris versions up to Solaris 7. (Note: The algorithm has been replaced in Solaris 8 by a new cache architecture, and priority paging should not be enabled on Solaris 8.) It was not enabled by default in Solaris 7 only because it was introduced very late in the Solaris release cycle.

- Page Scanner CPU Utilization Clamp

  Solaris internals PP.185-186

A CPU utilization clamp on the scan rate prevents the page-out daemon from using too much processor time. Two internal limits govern the desired and maximum CPU time that the scanner should use. Two parameters, `min_percent_cpu` and `max_percent_cpu`, govern the amount of CPU that the scanner can use. Like the scan rate, the actual amount of CPU that can be used at any given time is interpolated by the amount of free memory. It ranges from `min_percent_cpu` when free memory is at `lotsfree` (`cachefree` with priority paging enabled) to `max_percent_cpu` if free memory were to fall to zero. The defaults for `min_percent_cpu` and `max_percent_cpu` are 4% and 80% of a single CPU, respectively (the scanner is single threaded).

- Parameters That Limit Pages Paged Out

  Solaris internals P.186

Another parameter, `maxpgio`, limits the rate at which I/O is queued to the swap devices. It is set low to prevent saturation of the swap devices. The parameter defaults to 40 I/Os per second on sun4c, sun4m, and sun4u architectures and to 60 I/Os per second on the sun4d architecture. The default setting is often inadequate for modern systems and should be set to 100 times the number of swap spindles.

Because the page-out daemon also pages out dirty file system pages that it finds during scanning, this parameter can also indirectly limit file system throughput. File system I/O requests are normally queued and written by user processes and hence are not subject to `maxpgio`. However, when a lot of file system write activity is going on and many dirty file system pages are in memory, the page-out scanner trips over these and queues these I/Os; as a result, the `maxpgio` limit can sometimes affect file system write throughput. Please refer to the memory parameter appendix for further recommendations.

- Page Scanner Implementation

  Solaris internals PP.187-189

The page scanner is implemented as two kernel threads, both of which use process number 2,

"pageout." One thread scans pages, and the other thread pushes the dirty pages queued for I/O to the swap device. In addition, the kernel callout mechanism wakes the page scanner thread when memory is insufficient. (The kernel callout scheduling mechanism is discussed in detail in Section 2.5, "The Kernel Callout Table," on page 47.)

The scanner `schedpaging()` function is called four times per second by a callout placed in the callout table. The `schedpaging()` function checks whether free memory is below the threshold (`lotsfree` or `cachefree`) and, if required, prepares to trigger the scanner thread. The page scanner is not only awakened by the callout thread, it is also triggered by the `clock()` thread if memory falls below `minfree` or by the page allocator if memory falls below `throttlefree`.

Figure 5.25 illustrates how the page scanner works.

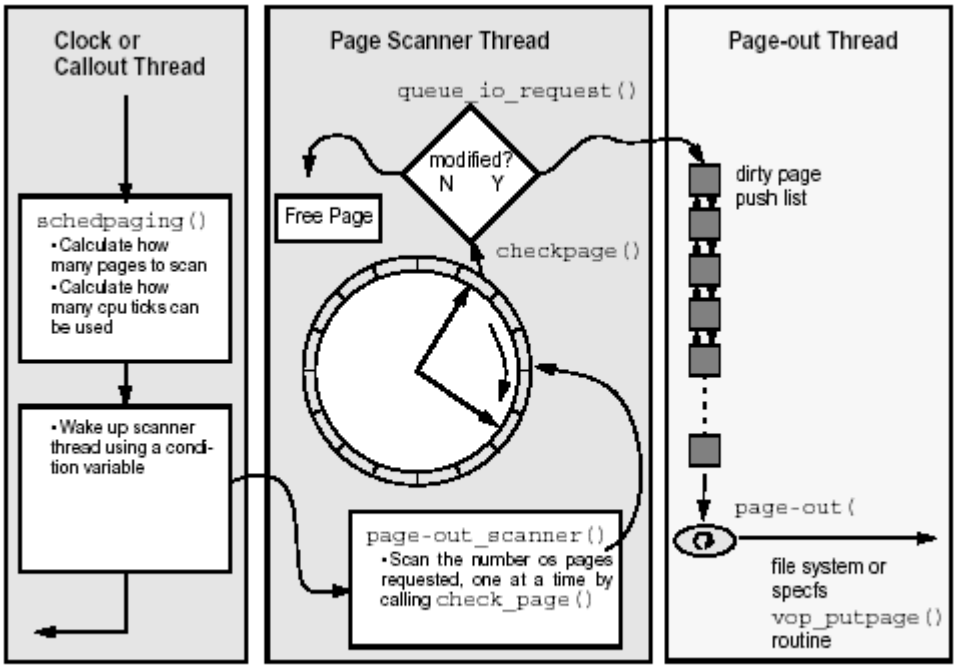

Figure 5.25 *Page Scanner Architecture*

When called, the `schedpaging` routine calculates two setup parameters for the page scanner thread: the number of pages to scan and the number of CPU ticks that the scanner thread can consume while doing so. The number of pages and cpu ticks are calculated according to the equations shown in "Scan Rate Parameters (Assuming No Priority Paging)" on page 180 and "Page Scanner CPU Utilization Clamp" on page 185. Once the scanning parameters have been calculated, `schedpaging` triggers the page scanner through a condition variable `wakeup`.

The page scanner thread cycles through the physical page list, progressing by the number of pages requested each time it is woken up. The front hand and the back hand each have a page pointer. The front hand is incremented first so that it can clear the referenced and modified bits for the page currently pointed to by the front hand. The back hand is then incremented, and the status of the page pointed to by the back hand is checked by the `check_page()` function. At this point,

if the page has been modified, it is placed in the dirty page queue for processing by the page-out thread. If the page was not referenced (it's clean!), then it is simply freed.

Dirty pages are placed onto a queue so that a separate thread, the page-out thread, can write them out to their backing store. We use another thread so that a deadlock can't occur while the system is waiting to swap a page out. The page-out thread uses a preinitialized list of async buffer headers as the queue for I/O requests. The list is initialized with 256 entries, which means the queue can contain at most 256 entries. The number of entries preconfigured on the list is controlled by the `async_request_size` system parameter. Requests to queue more I/Os onto the queue will be blocked if the entire queue is full (256 entries) or if the rate of pages queued has exceeded the system maximum set by the `maxpgio` parameter.

The page-out thread simply removes I/O entries from the queue and initiates I/O on it by calling the `vnode putpage()` function for the page in question. In the Solaris kernel, this function calls the `swapfs_putpage()` function to initiate the swap page-out via the `swapfs` layer. The `swapfs` layer delays and gathers together pages (16 pages on sun4u), then writes these out together. The `klustsize` parameter controls the number of pages that `swapfs` will cluster; the defaults are shown in Table 5-14. (See "The swapfs Layer" on page 156.)

Table 5-14 `swapfs` *Cluster Sizes*

Platform	Number of Clustered Pages (set by klustsize)
sun4u	16 (128k)
sun4m	31 (124k)
sun4d	31 (124k)
sun4c	31 (124k)
i86	14 (56k)

- The Memory Scheduler

Solaris internals P.189

In addition to the page-out process, the CPU scheduler/dispatcher can swap out entire processes to conserve memory. This operation is separate from page-out. Swapping out a process involves removing all of a process's thread structures and private pages from memory, and setting flags in the process table to indicate that this process has been swapped out. This is an inexpensive way to conserve memory, but it dramatically affects a process's performance and hence is used only when paging fails to consistently free enough memory.

The memory scheduler is launched at boot time and does nothing unless memory is consistently less than `desfree` memory (30 second average). At this point, the memory scheduler starts looking for processes that it can completely swap out. The memory scheduler will soft-swap out processes if the shortage is minimal or hard-swap out processes in the case of a larger memory shortage.

- Soft Swapping

Solaris internals PP.189-190

Soft swapping takes place when the 30-second average for free memory is below `desfree`. Then, the memory scheduler looks for processes that have been inactive for at least `maxslp` seconds. When the memory scheduler find a process that has been sleeping for `maxslp` seconds, it swaps out the thread structures for each thread, then pages out all of the private pages of memory for that process.

- Hard Swapping

Solaris internals P.190

Hard swapping takes place when all of the following are true:

- ✧ At least two processes are on the run queue, waiting for CPU.
- ✧ The average free memory over 30 seconds is consistently less than desfree.
- ✧ Excessive paging (determined to be true if page-out + page-in > maxpgio) is going on.

When hard swapping is invoked, a much more aggressive approach is used to find memory. First, the kernel is requested to unload all modules and cache memory that are not currently active, then processes are sequentially swapped out until the desired amount of free memory is returned. Parameters that affect the Memory Scheduler are shown in Table 5-15.

Table 5-15 *Memory Scheduler Parameters*

Parameter	Affect on Memory Scheduler
`desfree`	If the average amount of free memory falls below `desfree` for 30 seconds, then the memory scheduler is invoked.
`maxslp`	When soft-swapping, the memory scheduler starts swapping processes that have slept for at least `maxslp` seconds. The default for `maxslp` is 20 seconds and is tunable.
`maxpgio`	When the run queue is greater than 2, free memory is below `desfree`, and the paging rate is greater than `maxpgio`, then hard swapping occurs, unloading kernel modules and process memory.