NAME

tuning - performance tuning under FreeBSD

SYSTEM SETUP - DISKLABEL, NEWFS, TUNEFS, SWAP

The swap partition should typically be approximately 2x the size of main memory for systems with less than 4GB of RAM, or approximately equal to the size of main memory if you have more. Keep in mind future memory expansion when sizing the swap partition. Configuring too little swap can lead to inefficiencies in the VM page scanning code as well as create issues later on if you add more memory to your machine. On larger systems with multiple disks, configure swap on each drive. The swap partitions on the drives should be approximately the same size. The kernel can handle arbitrary sizes but internal data structures scale to 4 times the largest swap partition. Keeping the swap partitions near the same size will allow the kernel to optimally stripe swap space across the N disks. Do not worry about overdoing it a little, swap space is the saving grace of UNIX and even if you do not normally use much swap, it can give you more time to recover from a runaway program before being forced to reboot.

It is not a good idea to make one large partition. First, each partition has different operational characteristics and separating them allows the file system to tune itself to those characteristics. For example, the root and */usr* partitions are read-mostly, with very little writing, while a lot of reading and writing could occur in */var/tmp*. By properly partitioning your system fragmentation introduced in the smaller more heavily write-loaded partitions will not bleed over into the mostly-read partitions.

Properly partitioning your system also allows you to tune newfs(8), and tunefs(8) parameters. The only tunefs(8) option worthwhile turning on is *softupdates* with "tunefs -n enable /filesystem". Softupdates drastically improves meta-data performance, mainly file creation and deletion. We recommend enabling softupdates on most file systems; however, there are two limitations to softupdates that you should be aware of when determining whether to use it on a file system. First, softupdates guarantees file system consistency in the case of a crash but could very easily be several seconds (even a minute!) behind on pending write to the physical disk. If you crash you may lose more work than otherwise. Secondly, softupdates delays the freeing of file system blocks. If you have a file system (such as the root file system) which is close to full, doing a major update of it, e.g., "make installworld", can run it out of space and cause the update to fail. For this reason, softupdates will not be enabled on the root file system during a typical install. There is no loss of performance since the root file system is rarely written to.

A number of run-time mount(8) options exist that can help you tune the system. The most obvious and most dangerous one is **async**. Only use this option in conjunction with gjournal(8), as it is far too dangerous on a normal file system. A less dangerous and more useful mount(8) option is called **noatime**. UNIX file systems normally update the last-accessed time of a file or directory whenever it is accessed. This operation is handled in FreeBSD with a delayed write and normally does not create a burden on the system. However, if your system is accessing a huge number of files on a continuing

basis the buffer cache can wind up getting polluted with atime updates, creating a burden on the system. For example, if you are running a heavily loaded web site, or a news server with lots of readers, you might want to consider turning off atime updates on your larger partitions with this mount(8) option. However, you should not gratuitously turn off atime updates everywhere. For example, the */var* file system customarily holds mailboxes, and atime (in combination with mtime) is used to determine whether a mailbox has new mail. You might as well leave atime turned on for mostly read-only partitions such as */* and */usr* as well. This is especially useful for */* since some system utilities use the atime field for reporting.

STRIPING DISKS

In larger systems you can stripe partitions from several drives together to create a much larger overall partition. Striping can also improve the performance of a file system by splitting I/O operations across two or more disks. The gstripe(8), gvinum(8), and ccdconfig(8) utilities may be used to create simple striped file systems. Generally speaking, striping smaller partitions such as the root and */var/tmp*, or essentially read-only partitions such as */usr* is a complete waste of time. You should only stripe partitions that require serious I/O performance, typically */var*, */home*, or custom partitions used to hold databases and web pages. Choosing the proper stripe size is also important. File systems tend to store meta-data on power-of-2 boundaries and you usually want to reduce seeking rather than increase seeking. This means you want to use a large off-center stripe size such as 1152 sectors so sequential I/O does not seek both disks and so meta-data is distributed across both disks rather than concentrated on a single disk.

SYSCTL TUNING

sysctl(8) variables permit system behavior to be monitored and controlled at run-time. Some sysctls simply report on the behavior of the system; others allow the system behavior to be modified; some may be set at boot time using rc.conf(5), but most will be set via sysctl.conf(5). There are several hundred sysctls in the system, including many that appear to be candidates for tuning but actually are not. In this document we will only cover the ones that have the greatest effect on the system.

The *vm.overcommit* sysctl defines the overcommit behaviour of the vm subsystem. The virtual memory system always does accounting of the swap space reservation, both total for system and per-user. Corresponding values are available through sysctl *vm.swap_total*, that gives the total bytes available for swapping, and *vm.swap_reserved*, that gives number of bytes that may be needed to back all currently allocated anonymous memory.

Setting bit 0 of the *vm.overcommit* sysctl causes the virtual memory system to return failure to the process when allocation of memory causes *vm.swap_reserved* to exceed *vm.swap_total*. Bit 1 of the sysctl enforces RLIMIT_SWAP limit (see getrlimit(2)). Root is exempt from this limit. Bit 2 allows to count most of the physical memory as allocatable, except wired and free reserved pages (accounted by *vm.stats.vm.v_free_target* and *vm.stats.vm.v_wire_count* sysctls, respectively).

The *kern.ipc.maxpipekva* loader tunable is used to set a hard limit on the amount of kernel address space allocated to mapping of pipe buffers. Use of the mapping allows the kernel to eliminate a copy of the data from writer address space into the kernel, directly copying the content of mapped buffer to the reader. Increasing this value to a higher setting, such as '25165824' might improve performance on systems where space for mapping pipe buffers is quickly exhausted. This exhaustion is not fatal; however, and it will only cause pipes to fall back to using double-copy.

The *kern.ipc.shm_use_phys* sysctl defaults to 0 (off) and may be set to 0 (off) or 1 (on). Setting this parameter to 1 will cause all System V shared memory segments to be mapped to unpageable physical RAM. This feature only has an effect if you are either (A) mapping small amounts of shared memory across many (hundreds) of processes, or (B) mapping large amounts of shared memory across any number of processes. This feature allows the kernel to remove a great deal of internal memory management page-tracking overhead at the cost of wiring the shared memory into core, making it unswappable.

The *vfs.vmiodirenable* sysctl defaults to 1 (on). This parameter controls how directories are cached by the system. Most directories are small and use but a single fragment (typically 2K) in the file system and even less (typically 512 bytes) in the buffer cache. However, when operating in the default mode the buffer cache will only cache a fixed number of directories even if you have a huge amount of memory. Turning on this sysctl allows the buffer cache to use the VM Page Cache to cache the directories. The advantage is that all of memory is now available for caching directories. The disadvantage is that the minimum in-core memory used to cache a directory is the physical page size (typically 4K) rather than 512 bytes. We recommend turning this option off in memory-constrained environments; however, when on, it will substantially improve the performance of services that manipulate a large number of files. Such services can include web caches, large mail systems, and news systems. Turning on this option will generally not reduce performance even with the wasted memory but you should experiment to find out.

The *vfs.write_behind* sysctl defaults to 1 (on). This tells the file system to issue media writes as full clusters are collected, which typically occurs when writing large sequential files. The idea is to avoid saturating the buffer cache with dirty buffers when it would not benefit I/O performance. However, this may stall processes and under certain circumstances you may wish to turn it off.

The *vfs.hirunningspace* sysctl determines how much outstanding write I/O may be queued to disk controllers system-wide at any given time. It is used by the UFS file system. The default is self-tuned and usually sufficient but on machines with advanced controllers and lots of disks this may be tuned up to match what the controllers buffer. Configuring this setting to match tagged queuing capabilities of controllers or drives with average IO size used in production works best (for example: 16 MiB will use 128 tags with IO requests of 128 KiB). Note that setting too high a value (exceeding the buffer cache's write threshold) can lead to extremely bad clustering performance. Do not set this value arbitrarily high!

Higher write queuing values may also add latency to reads occurring at the same time.

The *vfs.read_max* sysctl governs VFS read-ahead and is expressed as the number of blocks to pre-read if the heuristics algorithm decides that the reads are issued sequentially. It is used by the UFS, ext2fs and msdosfs file systems. With the default UFS block size of 32 KiB, a setting of 64 will allow speculatively reading up to 2 MiB. This setting may be increased to get around disk I/O latencies, especially where these latencies are large such as in virtual machine emulated environments. It may be tuned down in specific cases where the I/O load is such that read-ahead adversely affects performance or where system memory is really low.

The *vfs.ncsizefactor* sysctl defines how large VFS namecache may grow. The number of currently allocated entries in namecache is provided by *debug.numcache* sysctl and the condition debug.numcache < kern.maxvnodes * vfs.ncsizefactor is adhered to.

The *vfs.ncnegfactor* sysctl defines how many negative entries VFS namecache is allowed to create. The number of currently allocated negative entries is provided by *debug.numneg* sysctl and the condition vfs.ncnegfactor * debug.numneg < debug.numcache is adhered to.

There are various other buffer-cache and VM page cache related sysctls. We do not recommend modifying these values.

The *net.inet.tcp.sendspace* and *net.inet.tcp.recvspace* sysctls are of particular interest if you are running network intensive applications. They control the amount of send and receive buffer space allowed for any given TCP connection. The default sending buffer is 32K; the default receiving buffer is 64K. You can often improve bandwidth utilization by increasing the default at the cost of eating up more kernel memory for each connection. We do not recommend increasing the defaults if you are serving hundreds or thousands of simultaneous connections because it is possible to quickly run the system out of memory due to stalled connections building up. But if you need high bandwidth over a fewer number of connections, especially if you have gigabit Ethernet, increasing these defaults can make a huge difference. You can adjust the buffer size for incoming and outgoing data separately. For example, if your machine is primarily doing web serving you may want to decrease the recvspace in order to be able to increase the sendspace without eating too much kernel memory. Note that the routing table (see route(8)) can be used to introduce route-specific send and receive buffer size defaults.

As an additional management tool you can use pipes in your firewall rules (see ipfw(8)) to limit the bandwidth going to or from particular IP blocks or ports. For example, if you have a T1 you might want to limit your web traffic to 70% of the T1's bandwidth in order to leave the remainder available for mail and interactive use. Normally a heavily loaded web server will not introduce significant latencies into other services even if the network link is maxed out, but enforcing a limit can smooth things out and lead to longer term stability. Many people also enforce artificial bandwidth limitations in order to

ensure that they are not charged for using too much bandwidth.

Setting the send or receive TCP buffer to values larger than 65535 will result in a marginal performance improvement unless both hosts support the window scaling extension of the TCP protocol, which is controlled by the *net.inet.tcp.rfc1323* sysctl. These extensions should be enabled and the TCP buffer size should be set to a value larger than 65536 in order to obtain good performance from certain types of network links; specifically, gigabit WAN links and high-latency satellite links. RFC1323 support is enabled by default.

The *net.inet.tcp.always_keepalive* sysctl determines whether or not the TCP implementation should attempt to detect dead TCP connections by intermittently delivering "keepalives" on the connection. By default, this is enabled for all applications; by setting this sysctl to 0, only applications that specifically request keepalives will use them. In most environments, TCP keepalives will improve the management of system state by expiring dead TCP connections, particularly for systems serving dialup users who may not always terminate individual TCP connections before disconnecting from the network. However, in some environments, temporary network outages may be incorrectly identified as dead sessions, resulting in unexpectedly terminated TCP connections. In such environments, setting the sysctl to 0 may reduce the occurrence of TCP session disconnections.

The *net.inet.tcp.delayed_ack* TCP feature is largely misunderstood. Historically speaking, this feature was designed to allow the acknowledgement to transmitted data to be returned along with the response. For example, when you type over a remote shell, the acknowledgement to the character you send can be returned along with the data representing the echo of the character. With delayed acks turned off, the acknowledgement may be sent in its own packet, before the remote service has a chance to echo the data it just received. This same concept also applies to any interactive protocol (e.g., SMTP, WWW, POP3), and can cut the number of tiny packets flowing across the network in half. The FreeBSD delayed ACK implementation also follows the TCP protocol rule that at least every other packet be acknowledged even if the standard 40ms timeout has not yet passed. Normally the worst a delayed ACK can do is slightly delay the teardown of a connection, or slightly delay the ramp-up of a slow-start TCP connection. While we are not sure we believe that the several FAQs related to packages such as SAMBA and SQUID which advise turning off delayed acks may be referring to the slow-start issue.

The *net.inet.ip.portrange.** sysctls control the port number ranges automatically bound to TCP and UDP sockets. There are three ranges: a low range, a default range, and a high range, selectable via the IP_PORTRANGE setsockopt(2) call. Most network programs use the default range which is controlled by *net.inet.ip.portrange.first* and *net.inet.ip.portrange.last*, which default to 49152 and 65535, respectively. Bound port ranges are used for outgoing connections, and it is possible to run the system out of ports under certain circumstances. This most commonly occurs when you are running a heavily loaded web proxy. The port range is not an issue when running a server which handles mainly incoming connections, such as a normal web server, or has a limited number of outgoing connections, such as a

mail relay. For situations where you may run out of ports, we recommend decreasing *net.inet.ip.portrange.first* modestly. A range of 10000 to 30000 ports may be reasonable. You should also consider firewall effects when changing the port range. Some firewalls may block large ranges of ports (usually low-numbered ports) and expect systems to use higher ranges of ports for outgoing connections. By default *net.inet.ip.portrange.last* is set at the maximum allowable port number.

The *kern.ipc.soacceptqueue* sysctl limits the size of the listen queue for accepting new TCP connections. The default value of 128 is typically too low for robust handling of new connections in a heavily loaded web server environment. For such environments, we recommend increasing this value to 1024 or higher. The service daemon may itself limit the listen queue size (e.g., sendmail(8), apache) but will often have a directive in its configuration file to adjust the queue size up. Larger listen queues also do a better job of fending off denial of service attacks.

The *kern.maxfiles* sysctl determines how many open files the system supports. The default is typically a few thousand but you may need to bump this up to ten or twenty thousand if you are running databases or large descriptor-heavy daemons. The read-only *kern.openfiles* sysctl may be interrogated to determine the current number of open files on the system.

The *vm.swap_idle_enabled* sysctl is useful in large multi-user systems where you have lots of users entering and leaving the system and lots of idle processes. Such systems tend to generate a great deal of continuous pressure on free memory reserves. Turning this feature on and adjusting the swapout hysteresis (in idle seconds) via *vm.swap_idle_threshold1* and *vm.swap_idle_threshold2* allows you to depress the priority of pages associated with idle processes more quickly then the normal pageout algorithm. This gives a helping hand to the pageout daemon. Do not turn this option on unless you need it, because the tradeoff you are making is to essentially pre-page memory sooner rather than later, eating more swap and disk bandwidth. In a small system this option will have a detrimental effect but in a large system that is already doing moderate paging this option allows the VM system to stage whole processes into and out of memory more easily.

LOADER TUNABLES

Some aspects of the system behavior may not be tunable at runtime because memory allocations they perform must occur early in the boot process. To change loader tunables, you must set their values in loader.conf(5) and reboot the system.

kern.maxusers controls the scaling of a number of static system tables, including defaults for the maximum number of open files, sizing of network memory resources, etc. *kern.maxusers* is automatically sized at boot based on the amount of memory available in the system, and may be determined at run-time by inspecting the value of the read-only *kern.maxusers* sysctl.

The *kern.dfldsiz* and *kern.dflssiz* tunables set the default soft limits for process data and stack size

respectively. Processes may increase these up to the hard limits by calling setrlimit(2). The *kern.maxdsiz*, *kern.maxssiz*, and *kern.maxtsiz* tunables set the hard limits for process data, stack, and text size respectively; processes may not exceed these limits. The *kern.sgrowsiz* tunable controls how much the stack segment will grow when a process needs to allocate more stack.

kern.ipc.nmbclusters may be adjusted to increase the number of network mbufs the system is willing to allocate. Each cluster represents approximately 2K of memory, so a value of 1024 represents 2M of kernel memory reserved for network buffers. You can do a simple calculation to figure out how many you need. If you have a web server which maxes out at 1000 simultaneous connections, and each connection eats a 16K receive and 16K send buffer, you need approximately 32MB worth of network buffers to deal with it. A good rule of thumb is to multiply by 2, so 32MBx2 = 64MB/2K = 32768. So for this case you would want to set *kern.ipc.nmbclusters* to 32768. We recommend values between 1024 and 4096 for machines with moderates amount of memory, and between 4096 and 32768 for machines with greater amounts of memory. Under no circumstances should you specify an arbitrarily high value for this parameter, it could lead to a boot-time crash. The **-m** option to netstat(1) may be used to observe network cluster use.

More and more programs are using the sendfile(2) system call to transmit files over the network. The *kern.ipc.nsfbufs* sysctl controls the number of file system buffers sendfile(2) is allowed to use to perform its work. This parameter nominally scales with *kern.maxusers* so you should not need to modify this parameter except under extreme circumstances. See the *TUNING* section in the sendfile(2) manual page for details.

KERNEL CONFIG TUNING

There are a number of kernel options that you may have to fiddle with in a large-scale system. In order to change these options you need to be able to compile a new kernel from source. The config(8) manual page and the handbook are good starting points for learning how to do this. Generally the first thing you do when creating your own custom kernel is to strip out all the drivers and services you do not use. Removing things like INET6 and drivers you do not have will reduce the size of your kernel, sometimes by a megabyte or more, leaving more memory available for applications.

SCSI_DELAY may be used to reduce system boot times. The defaults are fairly high and can be responsible for 5+ seconds of delay in the boot process. Reducing SCSI_DELAY to something below 5 seconds could work (especially with modern drives).

There are a number of *_CPU options that can be commented out. If you only want the kernel to run on a Pentium class CPU, you can easily remove I486_CPU, but only remove I586_CPU if you are sure your CPU is being recognized as a Pentium II or better. Some clones may be recognized as a Pentium or even a 486 and not be able to boot without those options. If it works, great! The operating system will be able to better use higher-end CPU features for MMU, task switching, timebase, and even device

operations. Additionally, higher-end CPUs support 4MB MMU pages, which the kernel uses to map the kernel itself into memory, increasing its efficiency under heavy syscall loads.

CPU, MEMORY, DISK, NETWORK

The type of tuning you do depends heavily on where your system begins to bottleneck as load increases. If your system runs out of CPU (idle times are perpetually 0%) then you need to consider upgrading the CPU or perhaps you need to revisit the programs that are causing the load and try to optimize them. If your system is paging to swap a lot you need to consider adding more memory. If your system is saturating the disk you typically see high CPU idle times and total disk saturation. systat(1) can be used to monitor this. There are many solutions to saturated disks: increasing memory for caching, mirroring disks, distributing operations across several machines, and so forth.

Finally, you might run out of network suds. Optimize the network path as much as possible. For example, in firewall(7) we describe a firewall protecting internal hosts with a topology where the externally visible hosts are not routed through it. Most bottlenecks occur at the WAN link. If expanding the link is not an option it may be possible to use the dummynet(4) feature to implement peak shaving or other forms of traffic shaping to prevent the overloaded service (such as web services) from affecting other services (such as email), or vice versa. In home installations this could be used to give interactive traffic (your browser, ssh(1) logins) priority over services you export from your box (web services, email).

SEE ALSO

netstat(1), systat(1), sendfile(2), ata(4), dummynet(4), eventtimers(4), login.conf(5), rc.conf(5), sysctl.conf(5), ffs(7), firewall(7), hier(7), ports(7), boot(8), bsdinstall(8), ccdconfig(8), config(8), fsck(8), gjournal(8), gpart(8), gstripe(8), gvinum(8), ifconfig(8), ipfw(8), loader(8), mount(8), newfs(8), route(8), sysctl(8), tunefs(8)

HISTORY

The **tuning** manual page was originally written by Matthew Dillon and first appeared in FreeBSD 4.3, May 2001. The manual page was greatly modified by Eitan Adler <*eadler@FreeBSD.org*>.