# Running and tuning OpenBSD network servers in a production environment

Philipp Bühler
sysfive.com GmbH
pb@sysfive.com

Henning Brauer
BS Web Services
hb@bsws.de

October 8, 2002

## Abstract

Heavily loaded network servers can experience resource exhastion. At best, resource exhaustion will slow server response, but left uncorrected, it can result in a crash of the server.

In order to understand and prevent such situations, a knowledge of the internal operation of the operating system is required, especially how memory management works.

This paper will provide an understanding of the memory management of OpenBSD, how to monitor the current status of the system, why crashes occur and how to prevent them.

## 1 Motivation

Our main motivation for this paper was the lack of comprehensive documentation about tuning network servers running under OpenBSD [Ope02], especially with regard to the memory usage of the networking code in the kernel.

Either one can get general information, or is "left alone" with the source code. This paper outlines how to deal with these issues, without reading the source code. At least one does not need to start in "nowhere-land" and dig through virtually everything.

This paper aims to give a deeper understanding on how the kernel handles connections and interacts with userland applications like the Apache webserver.

## 2 Resource Exhaustions

Running a publicly accessible server can always lead to unexpected problems. Typically it happens that resources get exhausted. There are numerous reasons for this, including:

**Low Budget** There's not enough money to buy "enough" hardware which would run an untuned OS.

**Peaks** Overload situations which can be expected (e. g. special use) or not (e. g. getting "slashdotted").

**DoS** Denial-of-Service by attackers flooding the server.

No matter what reason leads to an exhaustion, there are also different types of resources which can suffer from such a situation. We briefly show common types and countermeasures. Afterwards we go into detail about memory exhaustion.

### 2.1 I/O Exhaustion

It's very typical for network servers to suffer in this area. Often people just add more CPU to "help" a slowly reacting server, but this wouldn't help in such a case.

Usually one can detect such an exhaustion by using `vmstat(8)` or `systat(8)`. Detailed usage is shown in Section 5.1 There are also numerous I/O "bottlenecks" possible, but one typical indication is the CPU being mostly idle and blocked processes waiting for resources. Further distinctions can be made:

### Disk

The process is waiting for blocks from (or to) the disk and cannot run on the CPU, even if the CPU is idle. This case could be resolved by moving from IDE to SCSI, and/or using RAID technology. If repetitive writes/reads are being done an increase of the filesystem-cache could also help [1]. Filesystem-cache can be configured with the kernel option `BUFCACHEPERCENT`[2].

### NIC

Choosing the right network card is important for busy servers. There are lots of low-end models like the whole Realtek range. These cards are relatively dumb themselves. On the other hand, there are chipsets with more intelligence. DEC's 21143, supported by the dc(4) driver, and Intel's newer chipsets, supported by the fxp(4) driver, have been proven to work well in high-load circumstances.

Low-end cards usually generate an interrupt for every packet received, which leads to the problems we describe in the next subsection. By using better cards, like the mentioned DEC and Intel ones, packets are getting combined, thus reducing the amount of interrupts.

Another important point is the physical media interface, e. g. `sqphy(4)`. Noise and distortion is a normal part of network communications, a good PHY will do a better job of extracting the data from the noise on the wire than a poor PHY will, reducing the number of network retransmissions required.

It might be a good idea to use Gigabit cards, even when running 100 MBit/s only. They are obviously built for much higher packet rates (and this is the real problem, not bandwidth) than FastEthernet ones, thus have more own intelligence and deal better with high loads.

[1] Though this has implications on the KVM, see the appropriate section

[2] for most kernel configurations, see `options(4)` and `config(8)`.

### IRQ

Every interrupt requires a context switch, from the process running when the IRQ took place, to the interrupt handler. As a number of things must be done upon entering the interrupt handler, a large quantity of interrupts can result in excess time required for context switching. One non-obvious way to reduce this load is to share interrupts between the network adapters, something permitted on the PCI bus. As many people are not even aware of the the possibility of interrupt sharing, and the benefits are not obvious, let's look at this a little closer.

With separate adapters on separate interrupt lines, when the first interrupt comes in, a context switch to the interrupt handler takes place. If another interrupt comes in from the other adapter while the first interrupt is still being handled, it will either interrupt the first handler, or be delayed until the first handler has completed, depending on priority, but regardless, two additional context switches will take place – one into the second handler, one back out.

In the case of the PCI and EISA busses, interrupts are level triggered, not edge triggered, which makes interrupt sharing possible. As long as the interrupt line is held active, a device needs servicing, even if the first device which triggered the interrupt has already been serviced. So, in this case, when the first adapter triggers the interrupt, there will be a context switch to the handler. Before the handler returns, it will see if any other devices need servicing, before doing a context switch back to the previous process.

In a busy environment, when many devices are needing service, saving these context switches can significantly improve performance by permitting the processor to spend more time processing data, rather than switching between tasks. In fact, in a very high load situation, it may be desireable to switch the adapters and drivers from an interrupt driven mode to a polling mode, though this is not supported on OpenBSD at this time.

### 2.2 CPU Exhaustion

Of course the CPU can be overloaded also while other resources are still fine. Besides buying more CPU power, which is not always possible, there are other ways to resolve this problem. Most common cases for this are:

**CGI** Excessive usage of CGI scripts, usually written in interpreter languages like PHP or Perl. Better (resource-wise) coding can help, as well as using modules like mod_perl[3] to reduce load.

**RDBM** Usually those CGI scrips use a database. Optimization of the connections and queries (Indexing, ..) is one way. There is also the complete offloading of the database to a different machine [4].

**SSL** Especially e-commerce systems or online banking sites suffer here. OpenBSD supports hardware-accelerators [5]. Typical cryptographic routines used for SSL/TLS can be offloaded to such cards in a transparent manner, thus freeing CPU time for processing requests.

## 3 Memory Exhaustion

Another case of overloading can be the exhaustion of memory resources. Also the speed of the allocator for memory areas has significant influence on the overall performance of the system.

### 3.1 Virtual Memory (VM)

VM is comprised of the physical RAM and possible swap space(s). Processes are loaded into this area and use it for their data structures. While the kernel doesn't really care about the current location of the process' memory space (or address space) it is recommended that especially the most active tasks (like the webserver application) never be swapped out or even subjected to paging.

With regard to reliability it's not critical if the amount of physical RAM is exhausted and heavy paging occurs, but performance-wise this should not happen. The paging could compete for Disk I/O with the server task, thus slowing down the general performance of the server. And, naturally, harddisks are slower than RAM by magnitudes.

It's most likely that countermeasures are taken after the server starts heavy paging, but it could happen that also the swap space, and thus the whole VM, is exhausted. If this occurs, sooner or later the machine will crash.

Even if one doesn't plan for the server starting to page out memory from RAM to swap, there should be some swap space. This prevents a direct crash, if the VM is exhausted. If swap is being used, one has to determine if this was a one-time-only peak, or if there is a general increase of usage on the paging server. In the latter case one should upgrade RAM as soon as possible.

In general it's good practice to monitor the VM usage, especially to track down when the swap space is being touched. See section 5 for details.

### 3.2 Kernel Virtual Memory (KVM)

Besides VM there is a reserved area solely for kernel tasks. On the common i386 architecture (IA-32) the virtual address space is 4GB. The OpenBSD/i386 kernel reserves 768MB since the 3.2 release (formerly 512MB) of this space for kernel structures, called KVM.

KVM is used for addressing the needs of managing any hardware in the system and small allocations[6] being needed by syscalls. The biggest chunks being used are the management of the VM (RAM and swap), filesystem-cache and storage of network buffers (mbuf).

Contrary to userland the kernel allocations can-

[3] This can have security implications, but this is another story.

[4] This could be unfeasible due to an already overloaded network or due to budget constraints.

[5] `crypto(4)`

[6] like pathname translations

not be paged out ("wired pages"). Actually it's possible to have pageable kernel memory, but this is rarely used (e. g. for pipe buffers) and not a concern in the current context. Thus, if the KVM is exhausted, the server will immediatly crash. Of course 768MB is the limit, but if there is less RAM available, this is the absolute limit for wired pages then. Non-interrupt-safe pages could be paged out, but this is a rare exception.

Since RAM has to be managed by kernel maps also, it's not wise to just upgrade RAM without need. More RAM leaves `less` space for other maps in KVM. Monitoring the "really" needed amount of RAM is recommended, if KVM exhaustions occur. For example, 128MB for a firewall is usually more than enough. Look at Section 7.2 for a typical hardware setup of a busy firewall.

This complete area is called `kernel_map` in the source and has several "submaps"[7]. One main reason for this is the locking of the address space. By this mapping other areas of the kernel can stay unlocked while another map is locked.
Main submaps are `kmem_map`, `pager_map`, `mb_map` and `exec_map`. The allocation is done at boot-time and is never freed, the size is either a compile-time or boot-time option to the kernel.

# 4 Resource Allocation

Since the exhaustion of KVM is the most critical situation one can encounter, we will now concentrate on how those memory areas are allocated.

Userland applications cannot allocate KVM needed for network routines directly. KVM is protected from userland processes completely, thus there have to be routines to pass data over this border. The userland can use a `syscall(2)` to accomplish that. For the case of networking the process would use `socket(2)` related calls, like `bind(2)`, `recv(2)`, etc.

Having this layer between userland and kernel, we will concentrate on how the kernel is allocating memory; the userland process has no direct influence on this. The indirect influence is the sending and receiving of data to or from the kernel by the userland process. For example the server handles a lot of incoming network data, which will fill up buffer space (mbufs) within the KVM. If the userland process is not handling this data fast enough, KVM could be exhausted. Of course the same is true if the process is sending data faster than the kernel can release it to the media, thus freeing KVM buffers.

[7] see /sys/uvm/uvm_km.c

## 4.1 mbuf

Historically, BSD uses `mbuf(9)`[8] routines to handle network related data. An mbuf is a data structure of fixed size of 256 bytes [9]. Since there is overhead for the mbuf header (m_hdr{}) itself, the payload is reduced by at least 20 bytes and up to 40 bytes[10].

Those additional 20 bytes overhead appear, if the requested data doesn't fit within two mbufs. In such a case an external buffer, called cluster, with a size of 2048 bytes[11], is allocated and referenced by the mbuf (m_ext{}).

Mbufs belonging to one payload packet are "chained" together by a pointer `mh_next`. `mh_nextpkt` points to the next chain, forming a queue of network data which can be processed by the kernel. The first member of such a chain has to be a "packet header" (mh_type M_PKTHDR).

Allocation of mbufs and clusters are obtained by macros (MGET, MCLGET, ..). Before the release of OpenBSD 3.0 those macros used `malloc(9)` to obtain memory resources.

If there were a call to MGET but no more space is left in the corresponding memory map, the kernel would panic[12].

[8] memory buffer
[9] defined by MSIZE.
[10] see `/usr/include/sys/mbuf.h` for details.
[11] defined by MCLBYTES
[12] "malloc: out of space in kmem_map"

## 4.2 pool

Nowadays OpenBSD uses `pool(9)` routines to allocate kernel memory. This system is designed for fast allocation (and freeing) of fixed-size structures, like mbufs.

There are several advantages in using `pool(9)` routines instead of the ones around `malloc(9)`:

- faster than malloc by caching constructed objects
- cache coloring (using offsets to more efficiently use processor cache with real-world hardware and programming techniques)
- avoids heavy fragmentation of available memory, thus wasting less of it
- provides watermarks and callbacks, giving feedback about pool usage over time
- only needs to be in kmem_map if used from interrupts
- can use different backend memory allocators per pool
- VM can reclaim free chunks before paging occurs, not more than to a limit (Maxpg) though

If userland applications are running on OpenBSD (> 3.0), `pool(9)` routines will be used automatically. But it's interesting for people who plan (or do so right now) to write own kernel routines where using `pool(9)` could gain significant performance improvements.

Additionally large chunks formerly in the kmem_map have been relocated to the kernel_map by using pools. Allocations for inodes, vnodes, .. have been removed from kmem_map, thus there is more space for mbufs, which need protection against interrupt reentrancy, if used for e. g. incoming network data from the NIC [13].

[13] kmem_map has to be protected by `splvm()`, see `spl(9)`.

# 5 Memory Measurement

Obviously one wants to know about memory exhaustion *before* it occurs. Additionally it can be of interest, which process or task is using memory. There are several tools provided in the base OpenBSD system for a rough monitoring of what is going on. For detailed analysis one has to be able to read and interpret the values provided by those tools, but sometimes one needs more details and can rely on 3rd party tools then.

Example outputs of the tools mentioned can be found in the Appendix.

## 5.1 Common tools

These are tools provided with OpenBSD, where some are rather well-known, but some are not. In any case, we have found that often the tools are used in a wrong fashion or the outputs are misinterpreted. It's quite important to understand what is printed out, even if it's a "known tool".

#### top

One of the most used tools is `top(1)`. It shows the current memory usage of the system. In detail one could see the following entries:

**Real:** `68M/117M act/tot`, where 68MB are currently used and another 49MB are allocated, but not currently used and may be subject to be freed.

**Free:** `3724K`, shows the amount of free physical RAM

**Swap:** `24M/256M used/tot`, 24MB of 256MB currently available swap space is used.

If one adds 3724kB to 117MB, the machine would have nearly 122MB RAM. This is, of course, not true. It has 128MB of RAM; the "missing" 6MB are used as filesystem-cache[14].

[14] dmesg: `using 1658 buffers containing 6791168 bytes (6632K) of memory`

Besides this rough look on the memory usage of the system, there are indicators for other resource exhaustions. In the line `CPU states:` there is an entry `x.y% interrupt`. See how to resolve high values, they slow down the performance.

Blocking disks can be detected in the `WAIT` column. For example an entry `getblk` shows that the process is waiting for data from a disk (or any other block device).

### ps

Another very common tool is `ps(1)` and it's related to `top(1)`. Where `top(1)` is usually used for an overview of the system, one can use `ps(1)` for detailed picking on the exact state of a process (or process group).

Additionally it can be closer to reality and the output is more flexible, thus one can do better post-processing in scripts or similar.

Probably most interesting are the options showing how much percentage CPU and VM a process is using. One can sort by CPU ('u') or VM usage ('v') to find a hogging process quickly.

### vmstat

`vmstat(8)` is the traditional "swiss army knife" for detailed looks on the systems current usage. It's perfect for a first glance on potential bottlenecks.

A vmstat-newbie will probably be baffled by the output, but with some experience it's rather easy to find out, what's happening and where potential problems are located.

The default output consists of six areas (procs, memory, page, disks, faults, cpu). Each areas has columns for related values:

**procs** `r b w`, shows how many processes are (r)unning, are being (b)locked or are (w)aiting. Blocked processes cannot change to running before the block is resolved, e. g. a process "hangs" in a `getblk` state and waits for disk I/O.
Waiting means that the process is ready to run, but has still not been scheduled, most likely because the CPU is overloaded with processes.

**memory** `avm fre`, number of pages (1024b) being allocated and on the free list. The `avm` value gives a better insight on the allocation, than the values from `top(1)`.

**page** `flt re at pi po fr sr`, page-in (pi) and page-out (po) are most interesting here. It indicates if, and how much, paging (or even swapping) occurs.

**disks** `sd0 cd0`, the columns here depend on the disk setup of course. Values are transfer per seconds on this device. If high values here correspond with blocked processes below `procs` this is a good indication that the disk subsystem could be too slow.

**faults** `in sys cs`, can indicate too many interrupts and context switches on the CPU. `sys` counts syscalls brought to the kernel, a rather hard value to interpret with regard to bottlenecks, but one can get an idea of how much traffic has to pass between userland and kernel for completing the task.

**cpu** `us sy id`, looked at separately not too informative, but in combination with other values it's one keypoint in figuring out the bottleneck. If processes are in 'w' state and 'id' is very low, a CPU exhaustion occurs. Processes being (b)locked and having high (id)le values detect I/O exhaustions. Having high (sy)stem values and (w)aiting and/or (b)locked processes indicate that the kernel is busy with itself too much; this is usually because of "bad" drivers. Compare to 'faults in' to find out if interrupts are killing the performance. If not it's still possible that the CPU is busy transfering blocks from disk devices, indicated by low disk transfers and blocked processes.

Already impressive diagnostic possibilities, but `vmstat(8)` can show even more interesting things.

Besides the options `-i` to show summaries about interrupt behaviour and `-s` to get information about the swap area, `vmstat -m` can provide a very detailed look on the current memory usage.

Like we already have shown OpenBSD uses `pool(9)` for network data, thus we concentrate now on the last chunk `vmstat -m` is reporting. Most interesting are the lines `mbpl` and `mclpl`, which represent the memory usage for mbufs (mbpl) and clusters (mclpl).

Interesting columns are `Size, Pgreq, Pgrel, Npage` and `Maxpg`. One can obtain the following information from that:

**Size** the size of a pool item

**Pgreq** reports how many pages have ever been allocated by this pool.

**Pgrel** the pool freed those pages to the system.

**Npage** currently allocated/used pages by the pool.

**Maxpg** maximum number of pages the pool can use, even if paging would occur. More precise: the pool can grow over this limit, but the pagedaemon can reclaim free pages being over this limit, if VM is running low.

#### netstat

Usually `netstat(1)` is used for gathering network configurations, but it also provides information about different memory usages.

`netstat -f inet`[15] shows information about current network activity. With regard to memory consumption the columns `Recv-Q` and `Send-Q` are of major interest.

Typically one will encounter entries in Send-Q for a busy webserver with a good network connection. Clients usually have significant smaller bandwith, thus the provided data of the webserver application cannot "leave" the system. It's getting queued on the network stack, eating up mbuf clusters.

Pending requests will show up in Recv-Q, indicating that the userland cannot process the data as fast as it is coming in over the network.

[15] or -f inet6

The latter case should be resolved, even if memory is not running low, since the system would appear sluggish to the client, which is usually not appreciated (by the admin and/or client).

In addition to `vmstat -m`, `netstat -m` can report further values about current mbuf and cluster usage. Most notably it reports how much memory is "really" used. `vmstat -m` shows how many pool items are allocated, but `netstat -m` then reports how many pool items are actually filled with data to be processed.

In fact one could calculate this in `vmstat -m` by substracting `Releases` from `Requests`, but with numbers like 10599250 and 10599245, this is not really practical. Another pitfall is that `vmstat -m` reports memory pages, where `netstat -m` reports pool items[16] used, despite its output of `mapped pages in use`.

Furthermore it splits up what type of, and how many, mbufs are used (packet headers, sockets, data, ..), and it gives a summary about how much memory is needed by the network stack, which would be rather tedious to calculate from the `vmstat -m` output.

#### systat

This tool provides a `top(1)` like display of information the previous tools would provide. Especially `systat vmstat` is a perfect overview about load, disk usage, interrupts, CPU and VM usage.

One can monitor the system in intervals, or collect the information over time.

### 5.2 Special tools

Besides the tools we have shown so far, there are additional possibilities to monitor the system. *symon* and *pftop* are in the ports collection. *KVMspy* is not even published for now, but it shows that it's possible to write own tools for specific monitorings without enormous effort[17].

[16] usually a factor of two.

[17] the source code is below 300 lines.

### symon

For monitoring overall resource usage over time frames, *symon* [Dij02] is a perfect tool. It queries the kernel via `sysctl` about common resources. It uses `rrdtool` [Oet02] as data storage backend. There is a data collector daemon, called `symon`, which runs on every monitored machine, sending the collected data to `symux`, usually running on a central machine, which stores them on disk. Additionally there is a web-interface, `symon-web`, providing graphical representation of the collected data.

After machines have been set up with detailed analysis, this output is enough to detect high-load situations and trigger countermeasures before exhaustion occurs.

If one wants a long-term analysis of detailed data, it's relativly easy to extend this tool. Symon is pretty new and under active development by Willem Dijkstra, but already very useful.

### pftop

If one wants to monitor specific areas, like `pf(4)`, *pftop* [Aca02] is a curses-based, real-time monitoring application providing that.

One can consider it as a netstat-variant, providing similar information, about the paket filter.

### KVMspy

For the absolute curious one, there will be *KVMspy*. Currently it shows a bit more (offsets) information than `vmstat -m` about pools and a bit less (only current and highwater).

But, for the interested hacker, this is maybe better example code how to poll the kernel states via `kvm(3)` routines. Queries via `sysctl(3)` can be found in symon or are added to KVMspy in the future.

# 6 Countermeasures

And finally we come to the interesting pieces. Several ways to determine where a lack of KVM resources occurs have been shown. So, what to do if it actually happens?

There are three important kernel options defining the KVM layout with regard to networking. `NMBCLUSTERS` and `NKMEMPAGES` are compile-time options, but can be set via `config(8)` as well. `MAX_KMAPENT` can only be set at compile-time.

## 6.1 NMBCLUSTERS

The maximum number of clusters for network data can be defined here. Naturally, it's difficult to calculate this value in advance. Most tuning guides recommend a value of 8192 here. We usually use this value, too.

People tend to raise this value further, not knowing what implications this can have on the system. A value of 8192 potentially uses 16MB for `mb_map`: $8192 * 2048 = 16777216$ (MCLBYTES is usually 2048).

Since this is only a "pre-allocation" and not real usage in the first place, this value can be sane. On the other hand, if there are other problems with KVM, this value may be lowered.

Looking at real-life usage of busy webservers (see 7.1) the high watermark of `mclpl` is 524 (1048 clusters), thus even the default of 2048 clusters would be sufficient. This high watermark (Hiwat in `vmstat -m`) is also perfect to determine the `mclpl` size for load-balanced servers.

Imagine a Hiwat of 1000 on both machines. If one machine has to go out of service, due to a crash or simply hardware maintenance, a pool size of >4000 would ensure that the remaining machine doesn't run out of clusters. Remember that `vmstat -m` reports pages, not items, thus one has to calculate $1000*2*2$ for `NMBCLUSTERS`.

Additionally it's important to track why clusters are used in larger numbers. We have shown in 5.1/netstat that it is important to have a quick passing from the Recv-Q to the server application. It's a better idea to improve the application performance in this area, than increasing NMBCLUSTERS and let the data sit in KVM. At least a rather empty Recv-Q leaves more space for the Send-Q, which cannot be influenced directly to free clusters.

After all, it's dangerous to use high-values for this (and the following) options without very detailed knowledge about what is happening in the kernel. A "just to be safe" tuning can easily lead to an unstable machine. We have seen people using a value of 65535 for NMBCLUSTERS, rendering a pre-allocation of 128MB – not a good idea and usually it doesn't gain anything, except problems. Think twice about those values.

### 6.2 NKMEMPAGES

This option defines the total size of kmem_map. Since this is not exclusively used for networking data, it is a bit difficult to calculate the value for this option.

Since kmem_map was freed from other usage (4.2) and the introduction of pool(9) ensures that there is more space here for mbufs anyway, so an exhaustion of kmem_map is less likely than before.

Tracking of the usage is still possible, though. Looking again at
tt vmstat -m, this time at mbpl, one can see a correlation between mbpl and mclpl. It's common that the page value is usually half (or less) the value from mclpl. Yet again, one has to take care of "items vs page-size". Mbufs are way smaller then a cluster, thus 16 mbufs fit in one page of memory.

A network connection using clusters needs at least two mbufs, one for the paket header and one for the reference to the cluster. Since not every connection uses clusters it's sane to assume that a value for NKMEMPAGES being twice the value of NMBCLUSTERS is a good starting point.

Again, one should raise this value very carefully. Blindly changing these values can introduce more problems, than are solved.

Additionally, if the option is not touched, the kernel gets a sane default value for NKMEMPAGES at compile-time, based on RAM available in the system. If the kernel is compiled on a different machine with a different amount of RAM, this option should be used. A typical calculation value is 8162 for a machine with 128MB of RAM; this can be determined by `sysctl -n vm.nkmempages`.

### 6.3 MAX_KMAPENT

Definition of the number of static entries in kmem_map. Like NKMEMPAGES, the value is calculated at compile-time if unset. The default of 1000 (at least, it is based on "maxusers") is usually enough.

Raising this value is discouraged, but could be needed, if panics (`uvm_mapent_alloc: out of static map entries ..`) occur. Usually this happens if kmem_map is highly fragmented, for example by a lot of small allocations.

## 7 Real-life Examples

Putting everything together, we provide two examples of machines running OpenBSD under high load. It shows that a careful kernel configuration and hardware selection has great influence on the performance and reliability.

### 7.1 chat4free.de Webserver

This machine, hosted by BSWS, is running the webserver for one of Germany's biggest chat systems, chat4free.de.

The site consists of static pages and public forums. The unusual problem here is the both the overall load and the enormous peaks which happen when numbers of users are disconnected from the chat server due to external network problems or crashes of the server itself. Unlike many web applications, this server has a huge volume of small packets, which demonstrates that loading is more an issue of users and packet counts than raw data transfer.

Originally, it was running one Apache instance for the entire application, on an 700MHz Athlon system with 1.5G RAM, running a highly modified OpenBSD 3.0. Unfortunately, this system sometimes crashed due to KVM exhaustion.

To address this problem, the system was switched to a new system, again an 700MHz Athlon with 512M RAM, running two Apache instances in chroot jails, on a fairly stock OpenBSD 3.1 system. The system has a network adapter based on a DEC/Intel 21143, with a Seeq 84220 PHY, and runs "headless" with a serial console.

One of the two Apache instances is stripped down as much as I could make it, and serves the static pages. This httpd binary is only 303k in size, compared to the almost 600k of the stock Apache. The second instance of Apache is much bigger, as it has PHP compiled in. I always use static httpds, rather than Dynamic Shared Objects (DSOs).

The kernel configuration is fairly stock. All unused hardware support and emulations are removed, option DUMMY_NOPS is enabled. NMBCLUSTERS is bumped to 8192, NKMEMPAGES to 16384. I considered raising MAX_KMAPENT from its default of 1000 to 1500 or so to be able to have even more concurrent Apache processes running, though there has been no real need yet in this application. The machine has an ordinary IDE hard disk for the system, content and logs are on a separate machine's RAID subsystem, mounted via NFS. Most static content ends up being cached, reducing network traffic.

The "lean" httpd instance is configured for up to 1000 concurrent httpd tasks, the "fat" one for up to 600. I've seen both reach their maximum limits at the same time, and the smaller machine handles this load without incident. This is due to the superior memory management in OpenBSD 3.1 and the smaller Apache configurations.

Detailed kernel configuration and dmesg(8) can be found in the Appendix.

### 7.2 A firewall at BSWS

One important fact about firewalling and filtering is that the bandwidth isn't the important issue, the issue is the packet rate (i.e., packets per second). Each packet needs to be handled by the network adapter, the TCP/IP stack and the filter, which each need to do roughly the same amount of work whether the packet is small or large.

The firewall that protects a number of the servers at BSWS is under rather heavy load, not really due to total bandwidth, but the large number of small packets involved. It is running on a 700MHz Duron with 128M RAM and three DEC/Intel 21143-based NICs (one is currently not in use). It boots from a small IDE hard disk, which is quite unimportant to this application.

The machine is running a highly customized version of OpenBSD. The base system is OpenBSD 3.0, but many pieces of what became OpenBSD 3.1 were imported, including 3.1's packet filter `pf(4)`. At the time this was put together, there was no other option for this application. Many of pf's newer features were needed, but it was not possible to wait for 3.1-Release, as the previous OpenBSD 2.9 firewall running IPFilter had saturated the processor at near 100% utilization at peak usage times, and delays were being noticed. The kernel configuration has had all uneeded hardware support and binary emulations removed, and the always famous NKMEMCLUSTERS=16384 and NMBCLUSTERS=8192 modifications. The number of VLAN interfaces was raised to 20 (from 2 in GENERIC).

As of October 5, the expanded ruleset has 1132 rules. The “quick” keyword is used in most places to reduce the number of rules that must be evaluated for each packet, otherwise the entire ruleset must be evaluated for each packet. The rules are ordered so that the ones I expect the most matches from are towards the top of the file. All pass rules keep state; not only is this good practice for security, but with pf, state table lookups are usually much faster than rule evaluation. No NAT takes place on this machine, only packet filtering.

On the external interface, there is only spoofing protection taking place. Incoming packets with a source IP of the internal networks, outgoing packets with an IP which is not from one of the internal networks, and all 127.0.0.0/8 traffic is dropped. Normally, one would also drop packets with RFC1918 ("private IP space"), however in this case, it is handled externally by the BSWS core routers, because there is valid traffic with RFC1918 IPs from other internal networks crossing this firewall.

The actual filtering policies are enforced on the inside (VLAN) interfaces, which has the benefit that packets attempting to cross between VLANs encounter the same rules as packets from the outside. Every packet passing the firewall is normalized using the scrub directive. OpenBSD 3.2 will support multiple scrub methods besides the classic buffering fragment cache. One of the more interesting is the crop method, which almost completely avoids buffering fragments.

The results have been impressive. In September, 2002, the state table reached a peak size of 29,390, with an average size of 11,000. Up to 15,330 state table lookups per second were performed with average of 5600. State table inserts and removals peaked at slightly over 200 per second each. The CPU load seldom exceeds 10%. Compare this to the old IPFilter solution running on the same hardware doing much the same task, where the CPU was maxed out with only 600 rules and a peak of 15,000 packets per second. pf has permitted considerable growth in the complexity of the rule sets and traffic, and as you can see, still leaves BSWS considerable room to grow. Since this firewall went into operation in March, 2002, there hasn't been a single problem with its hardware or software.

## 8 Conclusions

Running OpenBSD servers under high load is pretty safe nowadays. We have shown that the introduction of `pool(9)` made operation way better with regard to memory usage and performance.

We have shown how network traffic influences the memory usage of the kernel and how the pieces are related together.

The provided knowledge about monitoring a running system and potential countermeasures against resource exhaustions should help to deal with high-load situations better.

## 9 Acknowledgements

A big "thank you" goes to Nick Holland, who turned our crappy english into something useful and provided a lot of input on how to explain this difficult area better.

Thanks also to Artur Grabowski for implementing `pool(9)` in the OpenBSD kernel and for further explanations about KVM.

Several proof-readers helped on finding spelling errors and inconsistencies within the paper, a special thanks here for Daniel Lucq, who also wrote KVMspy.

And, of course, thanks to the OpenBSD developer team for working on a system which provides already sane defaults for operating a high-load server, and, not to forget, a very high level of security.

## References

[Aca02] Can E. Acar. Openbsd pf state viewer. http://www.eee.metu.edu.tr/~canacar/pftop/, 2002.

[Dij02] Willem Dijkstra. The small and secure active system monitor. http://www.xs4all.nl/~wpd/symon/, 2002.

[McK96] Marshall Kirk (et. al.) McKusick. *The design and implementation of the 4.4BSD operating system.* Addison-Wesley, 1996.

[Oet02] Tobi Oetiker. Round robin database. http://people.ee.ethz.ch/~oetiker/webtools/rrdtool/, 2002.

[Ope02] OpenBSD. http://www.openbsd.org/, 2002.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated, Vol. 2.* Addison-Wesley, 1994.

# A top

This machine is the main server of sysfive.com GmbH, slightly tuned it is really idle.

```
load averages:  0.19,  0.12,  0.09                                  14:19:57
68 processes:  1 running, 64 idle, 3 zombie
CPU states:  0.3% user,  0.9% nice,  0.3% system,  0.0% interrupt, 98.4% idle
Memory: Real: 49M/80M act/tot  Free: 41M  Swap: 0K/256M used/tot

  PID USERNAME PRI NICE  SIZE   RES STATE WAIT     TIME    CPU COMMAND
15902 root       2    0 2308K 1832K idle  select  19:39  0.00% isakmpd
27679 pb         2    0  964K 1468K sleep select   7:00  0.00% screen-3.9.11
19945 gowry      2    0 4644K 5096K idle  select   4:30  0.00% screen-3.9.11
 3605 postfix    2    0  304K  736K sleep select   4:29  0.00% qmgr
22360 root      18    0  640K 9944K sleep pause    2:53  0.00% ntpd
11827 pb         2    0  516K 1312K sleep poll     2:18  0.00% stunnel
[..]
```

# B ps

Same machine, same processes reported by `ps -axv`

```
USER       PID %CPU %MEM   VSZ   RSS TT  STAT  STARTED       TIME COMMAND
root     22360  0.0  7.6   640  9944 ??  Ss     8Aug02    2:48.24 ntpd -c /etc/ntp.conf
gowry    19945  0.0  3.9  4644  5096 ??  Ss     9Aug02    4:30.56 SCREEN (screen-3.9.11)
root     15902  0.0  1.4  2308  1832 ??  Is    31Jul02   19:39.33 isakmpd
pb       27679  0.0  1.1   964  1468 ??  Ss    13Jul02    6:59.75 SCREEN (screen-3.9.11)
pb       11827  0.0  1.0   516  1312 ??  Ss    13Jul02    2:15.55 stunnel
postfix   3605  0.0  0.6   304   736 ??  S      6Aug02    4:30.29 qmgr -l -t fifo -u
```

# C vmstat

Current vmstat output of the same machine (`vmstat 1 5`)

```
 procs   memory        page                       disks     faults     cpu
 r b w    avm    fre   flt  re  pi  po  fr  sr cd0 sd0   in    sy   cs us sy id
 1 0 0  50324  41608    14   0   0   0   0   0   0   1  234  7151  160  0  0 99
 0 0 0  50324  41608    10   0   0   0   0   0   0   0  233  1602  165  0  0 100
 0 0 0  50324  41608     6   0   0   0   0   0   0   0  233  1589  165  0  1 99
```

If the machine would have disk I/O blocking problems, the output could look like this. Note the idle CPU, but blocked processes are waiting for blocks from the busy drive.

```
 procs   memory        page                       disks     faults     cpu
 r b w    avm    fre   flt  re  pi  po  fr  sr cd0 sd0   in    sy   cs us sy id
 1 2 0  50324  41608    14   0   0   0   0   0   0 271  234  7151  160  1  3 96
 0 1 0  50324  41608    10   0   0   0   0   0   0 312  233  1602  165  0  4 96
 0 1 0  50324  41608     6   0   0   0   0   0   0 150  233  1589  165  0  2 98
```

Worst-case scenario, the machine does heavy paging, thus overloading the disk subsystem. Additionally the CPU is maxed out. Processes are waiting, interrupts cause massive context-switching. The values are arbitrary.

```
 procs   memory        page                    disks     faults     cpu
 r b w    avm    fre   flt  re  pi  po  fr  sr cd0 sd0   in    sy    cs us sy id
 1 2 1    324    608   314   0  25  35   0   0   0 271  412  7151 1931 80 19  1
 1 3 2    324    608   310   0  28  42   0   0   0 312  501  1602 1876 81 19  0
 1 2 1    324    608   306   0  21  38   0   0   0 150  467  1589 1911 85 12  3
```

Now let's have a look at the pool situation of a firewall. A nice example that the pool can grow over the initial limit (Maxpg 512, Hiwat 516), but somehow KVM is low, since a lot of requests are failing (Fail 14725). The kernel should be reconfigured with NMBCLUSTERS > 1024 (`vmstat -m | grep mclpl`).

```
Name        Size Requests Fail Releases Pgreq Pgrel Npage Hiwat Minpg Maxpg Idle
mclpl       2048  1758499 14725 1757480   518     2   516   516     4   512    4
```

## D netstat

All packet data is getting delivered to/from the sshd fast enough, so no queuing occurs.

```
Active Internet connections
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  172.23.1.1.22          10.172.2.32.1156       ESTABLISHED
tcp        0      0  172.23.1.1.22          172.23.1.3.39679       ESTABLISHED
tcp        0      0  172.23.1.1.22          192.168.1.5.42456      ESTABLISHED
```

Somehow either the uplink is saturated, or the remote clients are not retrieving data fast enough, thus the Send-Q is growing.

```
Active Internet connections
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0   5346  172.23.1.1.22          10.172.2.32.1156       ESTABLISHED
tcp        0      0  172.23.1.1.22          172.23.1.3.39679       ESTABLISHED
tcp        0   7159  172.23.1.1.22          192.168.1.5.42456      ESTABLISHED
```

For whatever reason, sshd is not processing data fast enough. Maybe the deciphering needs more CPU then available?

```
Active Internet connections
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp     8811      0  172.23.1.1.22          10.172.2.32.1156       ESTABLISHED
tcp     5820      0  172.23.1.1.22          172.23.1.3.39679       ESTABLISHED
tcp    11631      0  172.23.1.1.22          192.168.1.5.42456      ESTABLISHED
```

Let's have a look at the memory usage with `netstat -m`. The stack has to keep 85 clusters in KVM, somehow the application is processing data either too fast (Send-Q) or too slow (Recv-Q).

```
384 mbufs in use:
        100 mbufs allocated to data
        178 mbufs allocated to packet headers
        106 mbufs allocated to socket names and addresses
85/1048 mapped pages in use
3144 Kbytes allocated to network (8% in use)
0 requests for memory denied
0 requests for memory delayed
0 calls to protocol drain routines
```

## E systat

Looks like the machine is doing nothing? Wrong, look at the interrupt counting for dc0 and dc2. It's the BSWS' firewall described earlier.

```
    1 users    Load  0.05  0.08  0.08                  Sat Oct  5 17:22:05 2002

          memory totals (in KB)              PAGING   SWAPPING     Interrupts
         real   virtual    free              in  out   in  out     7903 total
Active  91472     93712   10848      ops                            100 clock
All    116216    118456  270684      pages                              pccom0
                                                                    128 rtc
Proc:r  d  s  w       Csw  Trp  Sys  Int  Sof  Flt        forks    3669 dc0
        1  9            6    5   21 7936    4    2        fkppw         dc1
                                                          fksvm         pciide0
   0.0% Sys   0.0% User   0.0% Nice  90.0% Idle           pwait    4006 dc2
|    |    |    |    |    |    |    |    |    |    |       relck
                                                          rlkok
                                                          noram
Namei         Sys-cache    Proc-cache    No-cache         ndcpy
    Calls     hits    %    hits     %    miss   %         fltcp
        2        2  100                                   zfod
                                                          cow
Discs  wd0                                            128 fmin
seeks                                                 170 ftarg
xfers                                                8446 itarg
Kbyte                                                  39 wired
  sec                                                     pdfre
                                                          pdscn
```

## F iostat

Medium, but constant, traffic on sd0. In fact I was generating traffic with `dd(1)`.

```
      tty            cd0             sd0             sd1             fd0             cpu
 tin tout  KB/t t/s MB/s   KB/t t/s MB/s   KB/t t/s MB/s   KB/t t/s MB/s  us ni sy in id
   0  540  0.00   0 0.00   0.50 2614 1.28   0.00   0 0.00   0.00   0 0.00   1  1  5  3 90
   0  179  0.00   0 0.00   0.50 2560 1.25   0.00   0 0.00   0.00   0 0.00   0  0  2  2 95
   0  344  0.00   0 0.00   0.50 2601 1.27   0.00   0 0.00   0.00   0 0.00   0  0  3  5 92
   0  181  0.00   0 0.00   0.50 2601 1.27   0.00   0 0.00   0.00   0 0.00   0  1  5  3 91
```

## G pftop

Easy and quick overview about current traffic filtering:

```
pfTop: Up State 1-3/64, View: default, Order: none

PR   DIR SRC                   DEST                   STATE   AGE   EXP  PKTS BYTES
icmp Out 192.168.100.32:361    192.168.100.22:361      0:0       9     1     2    96
icmp Out 192.168.100.32:361    192.168.100.23:361      0:0       9     1     2    96
tcp  In  192.168.100.7:1029    192.168.100.32:443      4:4    4165 86302 25871 9251K
```

## H KVMspy

The full output would be too long, thus shortened to relevant pools/maps. Somehow this machine is not really exhausted, even with the default settings.

```
_kmem_map @ 0xd0518cdc: total size = 33431552 bytes, [0xd0890000, 0xd2872000]
_kmem_map @ 0xd0518cdc: 103 entries, actual size = 2453504 bytes (7.34%)
_mb_map @ 0xd0890c00: total size = 4194304 bytes, [0xda63e000, 0xdaa3e000]
_mb_map @ 0xd0890c00: 5 entries, actual size = 118784 bytes (2.83%)
_socket_pool @ 0xd05424c8: currently has 6 pages (24576 bytes)
_socket_pool @ 0xd05424c8: high water mark of 12 pages (49152 bytes)
_nkmempages @ 0xd05029d4: 8162 (_nkmempages * PAGE_SIZE = 33431552 bytes)
_nmbclust @ 0xd04fb278: 2048 (_nmbclust * MCLBYTES = 4194304 bytes)
```

## I chat4free.de Webserver

I'm using a bit more aggressive timeouts on this machine to lower the number of concurrent connections. This inlcudes a shortened KeepAliveTimeout to 10 seconds in apache's config and the following addition to `/etc/sysctl.conf`:

```
net.inet.tcp.keepinittime=10
net.inet.tcp.keepidle=30
net.inet.tcp.keepintvl=30
net.inet.tcp.rstppslimit=400
net.inet.ip.redirect=0
net.inet.ip.maxqueue=1000
kern.somaxconn=256
```

The timeouts depend heavily on your usage profile and need to be tried. The above ones work fine here, and should fit for most well connected webservers.
dmesg:

```
OpenBSD 3.1 (windu) #0: Wed Apr 17 20:10:40 CEST 2002
    root@ozzel:/usr/src/sys/arch/i386/compile/windu
cpu0: AMD Athlon Model 4 (Thunderbird) ("AuthenticAMD" 686-class) 700 MHz
cpu0: FPU,V86,DE,PSE,TSC,MSR,PAE,MCE,CX8,APIC,SYS,MTRR,PGE,MCA,CMOV,PAT,PSE36,MMX,FXSR
real mem  = 536457216 (523884K)
avail mem = 494899200 (483300K)
```

```
using 5689 buffers containing 26927104 bytes (26296K) of memory
mainbus0 (root)
bios0 at mainbus0: AT/286+(86) BIOS, date 04/02/02, BIOS32 rev. 0 @ 0xfb210
apm0 at bios0: Power Management spec V1.2
apm0: AC on, battery charge unknown
pcibios0 at bios0: rev. 2.1 @ 0xf0000/0xb690
pcibios0: PCI IRQ Routing Table rev. 1.0 @ 0xfdbd0/176 (9 entries)
pcibios0: PCI Exclusive IRQs: 11
pcibios0: PCI Interrupt Router at 000:07:0 ("VIA VT82C596A PCI-ISA" rev 0x00)
pcibios0: PCI bus #1 is the last bus
pci0 at mainbus0 bus 0: configuration mode 1 (no bios)
pchb0 at pci0 dev 0 function 0 "VIA VT8363 Host" rev 0x03
ppb0 at pci0 dev 1 function 0 "VIA VT8363 PCI-AGP" rev 0x00
pci1 at ppb0 bus 1
pcib0 at pci0 dev 7 function 0 "VIA VT82C686 PCI-ISA" rev 0x40
pciide0 at pci0 dev 7 function 1 "VIA VT82C571 IDE" rev 0x06: ATA100, channel 0
 \configured to compatibility, channel 1 configured to compatibility
wd0 at pciide0 channel 0 drive 0: <IC35L060AVER07-0>
wd0: 16-sector PIO, LBA, 58644MB, 16383 cyl, 16 head, 63 sec, 120103200 sectors
wd0(pciide0:0:0): using PIO mode 4, Ultra-DMA mode 5
pchb1 at pci0 dev 7 function 4 "VIA VT82C686 SMBus" rev 0x40
dc0 at pci0 dev 8 function 0 "DEC 21142/3" rev 0x41: irq 11 address 00:00:cb:53:62:c3
sqphy0 at dc0 phy 17: Seeq 84220 10/100 media interface, rev. 0
isa0 at pcib0
isadma0 at isa0
pckbc0 at isa0 port 0x60/5
pckbd0 at pckbc0 (kbd slot)
pckbc0: using irq 1 for kbd slot
wskbd0 at pckbd0: console keyboard
pcppi0 at isa0 port 0x61
sysbeep0 at pcppi0
npx0 at isa0 port 0xf0/16: using exception 16
pccom0 at isa0 port 0x3f8/8 irq 4: ns16550a, 16 byte fifo
pccom0: console
pccom1 at isa0 port 0x2f8/8 irq 3: ns16550a, 16 byte fifo
biomask 4000 netmask 4800 ttymask 4802
pctr: user-level cycle counter enabled
mtrr: Pentium Pro MTRR support
dkcsum: wd0 matched BIOS disk 80
root on wd0a
rootdev=0x0 rrootdev=0x300 rawdev=0x302
```

Kernel config:

```
machine i386 # architecture, used by config; REQUIRED
option DIAGNOSTIC # internal consistency checks
option CRYPTO # Cryptographic framework
option SYSVMSG # System V-like message queues
option SYSVSEM # System V-like semaphores
option SYSVSHM # System V-like memory sharing
option FFS # UFS
option FFS_SOFTUPDATES # Soft updates
option QUOTA # UFS quotas
option MFS # memory file system
option TCP_SACK # Selective Acknowledgements for TCP
option NFSCLIENT # Network File System client
option NFSSERVER # Network File System server
option FIFO # FIFOs; RECOMMENDED
option KERNFS # /kern
option NULLFS # loopback file system
option UMAPFS # NULLFS + uid and gid remapping
option INET # IP + ICMP + TCP + UDP
option INET6 # IPv6 (needs INET)
option PULLDOWN_TEST # use m_pulldown for IPv6 packet parsing
pseudo-device pf 1 # packet filter
pseudo-device pflog 1 # pf log if
pseudo-device loop 2 # network loopback
```

```
pseudo-device bpfilter 8 # packet filter
pseudo-device vlan  2 # IEEE 802.1Q VLAN
pseudo-device pty 64 # pseudo-terminals
pseudo-device tb 1 # tablet line discipline
pseudo-device vnd 4 # paging to files
#pseudo-device ccd 4 # concatenated disk devices
pseudo-device ksyms 1 # kernel symbols device

option BOOT_CONFIG # add support for boot -c
option I686_CPU
option USER_PCICONF # user-space PCI configuration
option DUMMY_NOPS # speed hack; recommended
option COMPAT_LINUX # binary compatibility with Linux
option COMPAT_BSDOS # binary compatibility with BSD/OS

option  NMBCLUSTERS=8192
option  NKMEMPAGES=16384

maxusers 64 # estimated number of users
config bsd swap generic

mainbus0 at root
bios0 at mainbus0
apm0 at bios0 flags 0x0000 # flags 0x0101 to force protocol version 1.1
pcibios0 at bios0 flags 0x0000 # use 0x30 for a total verbose
isa0 at mainbus0
isa0 at pcib?
pci* at mainbus0 bus ?
option PCIVERBOSE
pchb* at pci? dev ? function ? # PCI-Host bridges
ppb* at pci? dev ? function ? # PCI-PCI bridges
pci* at ppb? bus ?
pci* at pchb? bus ?
pcib* at pci? dev ? function ? # PCI-ISA bridge
npx0 at isa? port 0xf0 irq 13 # math coprocessor
isadma0 at isa?
isapnp0 at isa?
option WSDISPLAY_COMPAT_USL # VT handling
option WSDISPLAY_COMPAT_RAWKBD # can get raw scancodes
option WSDISPLAY_DEFAULTSCREENS=6
option WSDISPLAY_COMPAT_PCVT # emulate some ioctls
pckbc0 at isa? # PC keyboard controller
pckbd* at pckbc? # PC keyboard
vga* at pci? dev ? function ?
wsdisplay* at vga? console ?
wskbd* at pckbd? console ?
pcppi0 at isa?
sysbeep0 at pcppi?
pccom0 at isa? port 0x3f8 irq 4 # standard PC serial ports
pccom1 at isa? port 0x2f8 irq 3
pciide* at pci ? dev ? function ? flags 0x0000
wd* at pciide? channel ? drive ? flags 0x0000
dc* at pci? dev ? function ? # 21143, "tulip" clone ethernet
sqphy* at mii? phy ? # Seeq 8x220 PHYs
pseudo-device pctr 1
pseudo-device mtrr 1 # Memory range attributes control
pseudo-device sequencer 1
pseudo-device wsmux 2
pseudo-device crypto 1
```