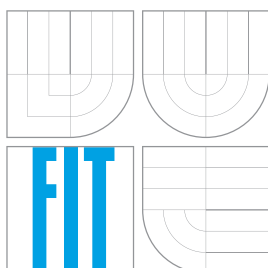


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

POKROČILÉ NÁSTROJE PRO MĚŘENÍ VÝKONU

ADVANCED TOOLS FOR PERFORMANCE MEASUREMENT

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE

AUTHOR

Bc. JAROMÍR SMRČEK

VEDOUCÍ PRÁCE

SUPERVISOR

Ing. TOMÁŠ KAŠPÁREK

BRNO 2008

Abstrakt

Tato práce prezentuje vstupně-výstupní vrstvu jádra Linux a ukazuje možnosti jejího ladění a optimalizace. Dále ukazuje nástroje, které je možno použít pro sledování systému a jejich výstupy. Práce se také soustřeďuje na kombinaci takových nástrojů, která by vedla k jednoduchému použití a komplexnímu výsledku sledování. Praktická část sestává z aplikace skriptů pro SystemTap a blktrace a z vlastního programu pro monitorování fragmentace s grafickým výstupem.

Klíčová slova

Linux, měření výkonu, I/O vrstva, bloková vrstva, kernel, optimalizace, výkon, NFS, CFQ plánovač, deadline plánovač, anticipatory plánovač, trasování I/O, blktrace, SystemTap, fragmentace, defragmentace, filefrag, server, zatížení

Abstract

This thesis presents the I/O layer of Linux kernel and shows various tools for tuning and optimization of its performance. Many tools are presented and their usage and outputs are studied. The thesis then focuses on the means of combining such tools to create more applicable methodology of system analysis and monitoring. The practical part consists of applying SystemTap scripts for blktrace subsystem and creating a fragmentation monitoring tool with graphical output.

Keywords

Linux, performance, performance measurement, I/O layer, block layer, kernel, optimization, NFS, CFQ scheduler, deadline scheduler, anticipatory scheduler, I/O tracing, blktrace, SystemTap, fragmentation, defragmentation, filefrag, server, workload

Citace

Jaromír Smrček: Advanced Tools for Performance Measurement, diplomová práce, Brno, FIT VUT v Brně, 2008

Advanced Tools for Performance Measurement

Prohlášení

Prohlašuji, že jsem tento diplomový projekt vypracoval samostatně pod vedením pana Ing. Tomáše Kašpárka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....

Jaromír Smrček
1. května 2008

© Jaromír Smrček, 2008.

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Contents

1	Introduction	4
1.1	Thesis structure	5
2	The I/O architecture of a Linux kernel	6
2.1	Device nodes	7
2.2	Accessing data on a block-device storage	7
2.3	I/O caching	7
2.4	Network devices	9
3	I/O Optimization	10
3.1	Choosing the I/O scheduler	10
3.2	Runtime parameters	11
3.2.1	Optimizing the I/O scheduler	11
3.2.2	Sysctl	12
3.3	Choosing the filesystem	13
3.4	System maintenance	14
3.5	Programming	14
4	Analysis and monitoring tools	15
4.1	Kernel messages	15
4.1.1	debugfs	15
4.2	Sysfs	16
4.3	Networking tools	16
4.3.1	ifconfig	16
4.3.2	iftop	16
4.3.3	ifstat	17
4.3.4	dstat	17
4.3.5	netstat	17
4.3.6	lsof	17
4.3.7	nfsstat	17
4.4	Disk tools	18
4.4.1	smartctl	18
4.4.2	lsof	18
4.4.3	iostat	18
4.4.4	dstat	18
4.4.5	filefrag	18
4.5	Kprobes	19
4.6	SystemTap	19

4.6.1	Implementation	20
4.6.2	Installing	20
4.6.3	Tracing using SystemTap	21
4.6.4	Tapsets	21
4.6.5	Safety	22
4.7	blktrace	22
4.7.1	Events	22
4.7.2	Output storing	25
5	Solving high I/O workload situations	26
5.1	Detecting the global source	26
5.1.1	CPU load in userspace	26
5.1.2	CPU load in kernel	27
5.1.3	Memory allocation in userspace	27
5.1.4	Memory allocation in kernel	28
5.1.5	Network bandwidth usage	28
5.1.6	Network latency	29
5.1.7	Disk I/O	29
5.2	Focusing on the disk I/O	30
5.2.1	Disk and partition	30
5.3	SystemTap scripts for blktrace	30
5.3.1	Parameters	31
5.3.2	countall.stp	31
5.3.3	spectest.stp	31
5.3.4	iotop.stp	32
5.3.5	topfile.stp	32
5.3.6	traceread.stp	32
5.4	Summary	32
6	Scenarios	34
6.1	Non-disk scenarios	34
6.1.1	High CPU load	34
6.1.2	High network usage	34
6.1.3	Too many network connections	35
6.1.4	Network protocol	35
6.1.5	Swapping	35
6.2	High disk usage	36
6.2.1	Dealing with disk-intensive processes	36
6.3	Badly positioned files	37
7	Fragmentation monitoring tool	39
7.1	Preliminaries	39
7.1.1	Fragmentation	39
7.1.2	Defragmentation	39
7.2	Motivation	39
7.3	Implementation	40
7.3.1	Output format	40
7.4	Sample output	41

7.5	Usage	41
8	Conclusion	46
8.1	Future work	47
	Bibliography	48
A	I/O tools outputs	50
A.1	ifstat	50
A.2	dstat	50
A.3	netstat	51
A.4	lsof	51
A.5	nfsstat	52
A.6	smartctl	53
A.7	iostat	54
A.8	blktrace	55
B	SystemTap examples	58
B.1	kprobeio.stp	58
B.2	countall.stp	58
B.3	spectest.stp	59
B.4	iotop.stp	59
B.5	topfile.stp	60
B.6	traceread.stp	60

Chapter 1

Introduction

Performance measurement of computers and computer systems is an integral part of system administration. Without the means of performance analysis, optimization and bottleneck elimination, there would be no possibility to ensure stable and up-to-date system performance.

In this thesis I focus on the performance measurement of the input-output (I/O) subsystem, more specifically the disk access time, throughput and load. This goal requires not only the study of block-layer subsystem but also networking, because disk access in modern computer systems is not only done locally.

The analysis of disk-based bottlenecks is not a simple nor sufficiently processed task at the time. There are many tools for monitoring, testing, benchmarking and tuning processors or memory access, but as for elaborate monitoring and data analysis of disk access, the situation is not that well. This fact is more significant in the way that disk is the slowest, therefore the weakest, link in the datapath.

Describing these problems, tools and techniques mentioned above is the first centerpiece of this thesis. The second one is using the knowledge about these tools to create a methodology that can be used in every-day system administration. Most disk-oriented analysis tools have all the important data, but do not show usable results for quick action or decision. My work will then focus on transformation of these outputs into an applicable form.

By presenting example situations and showing how given tools and methods should be applied to extract most information from a running system, I will try to give the administrator all the data needed to resolve situations in the shortest time possible.

As for more practical part of this work, specialized tool for fragmentation monitoring has been implemented and it is described in this thesis. Use of the tool should be helpful in many other aspects mentioned in the work.

Both investigative and practical parts of this thesis are based on the GNU/Linux operating system. Applying such procedures on other UNIX-based systems should not be too difficult, but as for Microsoft family of operating systems, the methodology of performance measurement and optimization is so different that supplying alternatives would be out of the scope of this thesis.

All tests and suggested settings have been tested on a dual-core machine running Gentoo GNU/Linux with a 2.6.23 kernel. Some paths or default settings can be distribution dependant, see your distribution handbook or documentation.

1.1 Thesis structure

The first chapter of this thesis is documenting the I/O architecture of the Linux kernel. After the global view it goes more deeply into the model of how the data on the disk is accessed through multiple layers and also specifies the network communication architecture. It also describes the disk datapath and all points of possible performance loss and/or optimization.

The second chapter presents the means of configuration and optimization of components to set the performance up and also create the basis for analysis and monitoring. Some configuration is done by kernel variables (and have to be enabled by compilation), others use runtime settings.

The third chapter sums up the tools used for performance measurement of studied subsystems used in the GNU/Linux environment. In addition to more or less known tools, I present more complex and profound tools that can be used for kernel data analysis and without which a sufficiently deep monitoring and analysis is impossible to be done in the I/O layer.

The fourth chapter is focused on using previously mentioned tools and giving procedural techniques for monitoring different areas on a running server. These methods are used to get useful information from the running system, using it to create countermeasures, optimizations and changes to prevent unwanted slowdown situations. Aside from methods using available tools, it also gives an overview on where to get the raw information for making an automated script or tool for this job. The chapter introduces specialized scripts for one of the advanced monitoring tools that are mostly suitable for disk I/O monitoring.

The fifth chapter tries to show the most usual situations that can happen on a running server (or a desktop system for the matter) and defines methods that can be applied to solve such situations. More specific information regarding the usage of given tools is also provided as a part of these solutions.

The sixth chapter takes some situations from the fifth chapter, but puts them into a more practical perspective and tries to solve them from the beginning. In contrast with the fifth chapter, where some methods have been defined, here they are applied in particular situations.

The last chapter presents the **fragmon** tool that has been implemented as a part of this thesis. Basic reasons and motivation for creating it and many aspects of implemented functionality are described. The tool can be used as a part of some problem-solving methods described earlier.

Chapter 2

The I/O architecture of a Linux kernel

To make a computer work properly, data paths must be provided to let the information flow between CPU(s), RAM, and multiple I/O devices that can be connected to a personal computer. These data paths, which are denoted as buses, act as primary communication channels inside the computer.

The data path that connects a CPU to an I/O device is generically called an I/O bus. Each device connected to the I/O bus has its own set of I/O addresses, which are usually called I/O ports. The x86 architecture uses 65,536 8-bit I/O ports which can be aggregated to create 16-bit or 32-bit ports. I/O ports may also be mapped into addresses of the physical address space instead of using specialized instructions (`in`, `out`) to access or write the data. In older kernels (2.4 and less) the mapped I/O could only be done in low memory and there had to be "bounce buffers" to use high memory which meant double buffering and slow performance.

Every device has its own set of registers and the program that is run on the computer must know the communicating protocol to communicate directly. Also peripherals are asynchronous mostly, so they use IRQ calls and DMA. Therefore to encapsulate the I/O protocol and access each device by the same means, device drivers are built and loaded into the kernel.

Device drivers provide the implementation of standard operations on a file (as the file is the basic notion of UNIX-type operating systems). Userspace application can use standard system calls (`write()`, `read()`, ...) to communicate with a peripheral device regardless of its I/O protocol. Device files are typically located in `/dev` as a convention.

Communication between the device driver and the device itself is mostly asynchronous, but system calls to pass data between the device through the kernel into the userspace process are done synchronously and system calls are usually blocking. There are ways to gain asynchronous access to the device via `aio` (see [2]).

2.1 Device nodes

Devices are identified by their major and minor number (the device type and the sequence identifier respectively) which can be passed to `mknod` command to create a device node (represented by a device file). Most of the devices today have their device files, with the main exception – network cards. Nowadays the creation of device nodes is done automatically via `udev` – userspace dev (see [1]).

Devices can be splitted into two main categories – character devices and block devices – based on the type of data communication. Character devices communicate by single characters (terminal, random number generator, ...), Block devices communicate by blocks of data (network card, disk, ...).

2.2 Accessing data on a block-device storage

Block devices are more sophisticated than character devices, not only the data is sent in a block, but there can be multiple levels of caching to create bigger blocks of data and increase efficiency of I/O communication. Many block devices can be connected to a single bus, so there has to be a fairness algorithm to select the appropriate device for communication. The hardware can have the access time for different data spanning over large interval and every device can have its own protocol.

The encapsulating layer for all the device drivers above the I/O protocols is called *generic block layer* and it creates I/O requests for adjacent blocks physically located on the disk device. It also brings other features to the kernel, like mapping and unmapping the data page frames only when needed by the CPU, zero-copy effort for passing data to the userspace (buffers from the userspace are used directly by DMA or CPU), logical volumes management (LVM, RAID) and advanced usage of modern controllers, DMA and caches.

Block devices can perform a one-block transaction, but it is a resource-wasting operation. Burst transfers are used instead. To improve performance, the request sent to the device can be rearranged in a way of maximizing throughput. A sub-layer of the generic block layer focusing on the rearrangement of requests for block devices is called *I/O scheduler* (more on I/O schedulers in 3.1).

Generic block layer is called by every filesystem module to access raw data. The filesystem module then transforms the data into inodes, files and directories for upper layers of the architecture. To encapsulate differences between multiple filesystems, a *virtual filesystem* layer provides unified interface for userspace processes.

2.3 I/O caching

Just like the CPU uses a fast SRAM cache to increase performance when accessing data in much slower DRAM memory, the I/O subsystem has a similar method of reducing disk access.

One level of caching is the *dentry cache* and *inode cache*, that keep the last few requested dentries and inodes (directory and file metadata) for further use. Many processes access the same file in the same directory multiple times, so caching the metadata (like location, size, etc.) causes a performance boost.

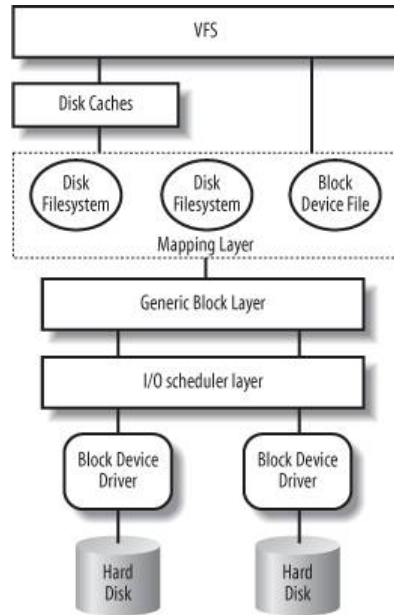


Figure 2.1: The basic schema of the datapath (from [3])

The main disk cache used in a Linux kernel is the *page cache*. It keeps whole pages of data (4kiB for x86 architectures) in memory instead of directly writing them to disk. This can be overridden by using direct I/O. The page is removed from the memory only when it is already written to the disk and there is no more space for new pages, otherwise the page stays in memory for further use. This applies for reading and also writing. When writing data to disk, page cache waits for more write requests before actually committing the changes – *deferred write*.

In order to use the page cache efficiently, it is stored as a radix-tree, where each node contains multiple information about the page itself. One of the most important information is the *dirty flag*, that specifies whether the page should be written to disk or not. Writing is deferred and occurs when the dirty page has been dirty for a long time, the page has been flushed or a process requests the writing explicitly (by `sync()` for example).

The page cache works on the level of *pages*, whereas the filesystem drivers work with *blocks* and the disk devices use *sectors* as a unit of data. Sectors are mostly 512B large (although there are few devices with 520, 1024 or 2048B sectors). The I/O scheduler and device driver must manage sectors of data. Block size is defined by each filesystem, but it has to be a power of 2, must contain an integral number of sectors and for some versions of filesystems (e.g. XFSv1) it must fit into a page (512, 1024, 2048 or 4096 bytes for x86).

This differentiation of data units means that every layer uses its own method of address translation and data caching. The process requests data from a file to be read to a memory buffer located in a certain page, the filesystem reads first block and puts it into the page. The block however usually consists of more than one sector that has to be read from a device. Devices today do not read only one sector, but read a *segment* instead (multiple adjacent sectors) and cache such data.

2.4 Network devices

Network interface cards (NICs) are block devices which on lower levels look much the same for the kernel. The difference is that there are no device nodes to create a unified entrypoints for processes. Instead a socket must be created, this socket is a file-descriptor bound to a certain network port and process. This differentiation from standard device files is given by the internet protocol architecture.

Linux network architecture is conform to the Internet model (not ISO/OSI model) of networking. The application layer resides in userspace and other layers in the kernel. Processes request/send data through a system call (`recv()` or `send()`) and it goes through all underlying layers (the transformation of TCP/UDP and IP packets is not significant for this thesis), till the encapsulated data is prepared to be sent using a link protocol. This is when the device driver takes over and sends the data through NIC.

Every NIC has two queues for data (one for sending and one for receiving) and uses IRQ mechanism to independently send and receive data. DMA is used for data communication between kernel and NIC.

Chapter 3

I/O Optimization

Optimizing the I/O subsystem on a GNU/Linux operating system can be done by changing kernel parameters (at boot or runtime), setting device driver parameters or using specialized tools or firmware commands. Most given optimizations are focused on the performance of the I/O layer, some optimizations can also speedup the access or reaction time for the device.

3.1 Choosing the I/O scheduler

Much like the process scheduler, the I/O scheduler distributes a shared resource among multiple processes. Unlike the process scheduler, the I/O scheduler is not mandatory, its purpose is solely to increase the performance by reducing disk access time.

When reading data from a disk, the slowest part of the process is to locate the requested block of data, this means "seeking" over the disk platter. On a multiuser system, many processes can request data located on different sectors, that are very far from each other, thus making the disk seek from one end of the disk to the other one over and over again.

The purpose of the I/O scheduler is to put all requests into a queue and sort it in a way that minimizes the disk seeking, thus reducing the average access time. The most important part of current schedulers is the elevator algorithm. There are four possible elevator algorithms in current kernels.

The name "elevator algorithm" comes from the idea of a basic scheduler used in older kernels, the Linux Elevator. The idea is an analogy to a real-life elevator, that goes from the lowest request floor to the topmost one and can stop on its way to pickup more people. For example requests <5 150 12 35 145 10> are handled this way: <5 10 12 35 145 150>.

There are flaws in the main idea, mainly there is a threat of starvation. This is where more sophisticated elevator algorithms come in.

- **NOOP** – Just a FIFO queue, no overhead, used when the scheduler is not wanted.
- **Deadline** – It has three different queues. Aside from the main FIFO queue, a pair of secondary queues (for read and write requests) is added to store the requests' timeout. After a timeout in secondary queue(s) the scheduler moves to a sector from such queue, dispatches timeouted requests and continues with the main queue.
- **Anticipatory** – Based on the deadline scheduler, but because processes usually request multiple sectors in a stream, it waits a small amount of time after the request anticipating another request on the same spot.

- **CFQ** – Completely fair queuing. Every process has its own elevator queue and the system handles them with a round-robin algorithm or by priority.

Test results for scheduler comparison can be found at [4], [5] or [6]. Results show that the deadline scheduler prevents starvation (e.g. by reading a big file in a stream), but still a stream of writes can starve reading. The anticipatory scheduler deals with both such problems and when the anticipation meets the real situation, it excels over other schedulers (but when the reads do not go in a stream, the anticipation is a slowdown).

Default scheduler for the linux kernel is currently the CFQ scheduler. The anticipatory scheduler can be used in smaller systems, but basically the CFQ is better overall. The CFQ scheduler has been chosen over the anticipatory scheduler because of its fairness in distributing the I/O requests over multiple devices and scalability per process.

There are however cases, when we do not want the I/O scheduler to rearrange I/O requests. This can be due to the nature of the device, where the access time is not based on seeking (SSD). Another use of the NOOP scheduler is for highly efficient disk drivers supporting their own reordering of commands (NCQ, TCQ, ...).

To provide I/O schedulers other than NOOP, compile kernel with options `CONFIG_IOSCHED_DEADLINE`, `CONFIG_IOSCHED_AS` and/or `CONFIG_IOSCHED_CFQ`. The current scheduler can be set by `CONFIG_DEFAULT_IOSCHED` or by kernel boot parameter `elevator=` or via `/sys/block/<disk>/queue/scheduler`.

3.2 Runtime parameters

3.2.1 Optimizing the I/O scheduler

The `sysfs` (see chapter 4.2) can be used to change parameters of the I/O scheduler. The scheduler is selected via `/sys/block/<disk>/queue/scheduler`. Based on the selected scheduler the `/sys/block/<disk>/queue/iosched/` directory is filled with files holding parameters of the current scheduler, that can be read or set by `cat` or `echo` command.

Changing the default values is not recommended, although in some special cases it can get better results (e.g. changing the readahead size or anticipation timeout), for complete description, see [6].

Because with CFQ every process has its own queue, processes can have priorities for disk access. The tool used for changing the priority of a process is called `ionice` (a part of `schedutils`). Usage is analogical to the `nice` tool (you can set either real-time priority, idle-priority or best-effort, in which case there are 8 levels of niceness).

Some parameters that optimize the CFQ scheduler (relative to `/sys/block/<disk>/`):

- `read_ahead_kb` – Sets the readahead buffer size, default is 128, 512 can speedup streaming reads.
- `nr_requests` – Sets queue length, default is 128, higher value speeds-up at the expense of latency.
- `iosched/back_seek_*` – Configure back seeking (like in the anticipatory scheduler), CFQ is an ascending elevator otherwise.
- `iosched/fifo_batch_*` – Configure expiration times for deadline scheduling in CFQ.

3.2.2 Sysctl

Some of the most notable performance improvements for Linux can be accomplished via system control (sysctl variables) in `/proc/sys`. Unlike most other areas of `/proc`, sysctl variables are typically writable and are used to adjust the running kernel rather than simply monitor currently running processes and system information.

There are two ways to work with sysctl: by directly reading and modifying files in `/proc/sys` or by using the `sysctl` command. Direct reading and modifying means the usage of `cat` and `echo` commands. These changes however are only temporary. To make the effect permanent, all changes must be done in `/etc/sysctl.conf` configuration file (location can be distribution dependant) which is loaded at startup.

The `sysctl` command uses the same hierarchy as the directory structure in `/proc/sys` where forward slash is replaced by a fullstop. For example:

```
# echo "1" > /proc/sys/net/ipv4/ip_forward
equals
# sysctl -w net.ipv4.ip_forward="1"
```

To reload the configuration file, simply type `sysctl -p`, to list available variables type `sysctl -a`.

Binary access to system information by `sysctl()` system call is now to be deprecated (it should have been removed in 2006 but then stayed in the kernel) and will be removed from Linux kernel in 2010. File-oriented access through `/proc/sys` should be used instead.

The most important values for I/O optimization in sysctl:

- **fs** – file systems – The directory structure depends on which filesystem you use. As for the general parameters, `file-nr` and `file-max` are relevant for this work. They specify how many handles are there and how many can be allocated.
- **net** – networking – This section offers the most options for increasing performance but it can also make the system non-compliant with other computers on the network. Among others, `iptables` can be configured here.
- **net.core** – General section, buffer setup, memory usage (`wmem_max`, `rmem_max`, `wmem_default`, `rmem_default`).
- **net.ipv4** – Overrides the core settings, setups the IPv4 protocol. Value `tcp_max_syn-backlog` specifies the maximum amount of half open connections and `tcp_syncookies` enables syn-cookies. These values can be increased (set on) for some webserver. `ip_local_port_range` is self-explanatory and by default set to 32768 61000. Buffers can be set more specifically by `tcp_wmem` `tcp_rmem` `tcp_mem`. Finally keepalive timeout can be reduced by `tcp_keepalive_time` from default 7200 to a lower value.
- **vm** – virtual memory – There are few variables that are very useful when tuning performance. They control the behavior when allocating, swapping or syncing memory.
- **vm.overcommit_memory** – By default it is set to 0, which means that the kernel heuristically estimates free memory left and prevents allocating more than this amount. When set to 1 kernel pretends that there is always enough memory, until it physically runs out of it. This can be used when running programs allocating large chunks of

memory and using only a little of it. Setting it to 2 uses the `vm.overcommit_ratio` which prevents allocating to exceed swap plus this percentage of physical memory.

- `vm.page-cluster` – On pagefault, the kernel loads not only one page, but $2^{\text{vm.pagecluster}}$ pages. Default is 3, maximum 5.
- `vm.dirty_ratio` – Percentage of total system memory pages at which dirty pages are written to disk. Per process.
- `vm.dirty_background_ratio` – Percentage of total system memory pages at which `pdflush` starts to write dirty data.
- `vm.dirty_writeback_centisecs` – Periodical timer to run `pdflush` and sync "old" data to disk. In 100's of seconds.
- `vm.dirty_expire_centisecs` – The age of data in 100's of seconds to be considered "old".
- `vm.laptop-mode` – Saves energy by minimizing spin-up times.

For full documentation of system control variables, see linux kernel documentation [21], Red Hat Magazine¹ (focused on virtual memory) or other community pages (like [8], [7]).

Some of the parameters can also be set by `hdparm` or `blockdev` tools.

3.3 Choosing the filesystem

The filesystem has a great impact on the system's performance. The overhead that can arise because of badly chosen filesystem can be overwhelming on a highly accessed disk.

For desktop computers, the filesystem choice is not such a big deal because the disk access is not very high nor constant and also those systems are more general-purpose oriented. Servers on the other hand have more specialized requirements for a disk. Some filesystems have better performance on many small files (like `ext2` or `ext3`), others on files with big amount of data (`XFS`, `JFS`).

Some filesystems can use native unicode in filenames (`JFS`, `NTFS`), some have short maximum filename length (`FAT32` without LFN) and the maximum file size also comes to question (`FAT32` - 4GiB, `ReiserFS 3.5` - 4GiB). The question of journalling is also important because it has a performance impact and some filesystems use full journalling while others only metadata journalling (nowadays most filesystems can turn full journalling on) or none at all (`ext2`). The choice then depends on the server's scope of operation.

Other properties like scalability or recovery possibilities can be compared. The `XFS` filesystem should not be used without uninterruptible power supply, because it doesn't synchronize with the disk very often (which is a great performance boost) and fills corrupted data files with zeroes on recovery. Also fragmentation is to be considered (`ReiserFS` gets fragmented very quickly).

Main parameters of mostly all known filesystems can be found on Wikipedia [9], one of the most complete comparisons for mainstream filesystems can be found at [10], [11]. The results in short show, that `ext2/ext3` are very good for great number of files, whereas `XFS` excels in big file handling and file deletion.

Setting the filesystem blocksize is essential for the balance between data access time for larger files and the amount of internal fragmentation when smaller files are stored.

¹<http://www.redhat.com/magazine/001nov04/features/vm/>

3.4 System maintenance

As in real life, the biggest enemy of a system in use is *time*. Every system should be properly maintained to prevent slowdowns. Checking hardware status of the server (error rates, SMART capabilities, etc.) and monitoring memory and disk usage status is essential for enterprise servers. The next few chapters will give more detailed information about system monitoring and available tools.

One problem that most definitely falls in this category of system optimization is fragmentation. Although linux filesystems do not fragment that much as many other known filesystems, fragmentation is still a problem. Also, more global scope of fragmentation is often omitted, the directory data fragmentation.

When files under one directory are far from each other, disk must seek more, thus creating a slowdown in the I/O layer. Monitoring and defragmenting essential directories should be done regularly.

3.5 Programming

Some of the performance can be gained by writing programs in optimized way (see [25]). Reading two subsequent small block instead of one larger can be slower because of the request queue at the disk. Random access is much slower than sequential, mapping into memory can be of a good use. If the application has its own caching system (like database servers usually do), direct I/O can be used (`open()` with `O_DIRECT`) so the OS caching is taken out.

There is also an asynchronous I/O project with limited functionality [2].

Chapter 4

Analysis and monitoring tools

Monitoring tools are not only needed when having performance difficulties. They are also useful for long-term logging, auditing, etc. This chapter gives an overview of many tools and their functionality. The choice of such tools depends on particular demands of administrators.

There are few very sophisticated and complex tools for overall system monitoring in current window-manager packages, like KSysGuard or gkrellm. All such applications are based on the same sources of data as the ones mentioned in next sections. The main purpose of these applications is to gather the information into one place and graphically represent it for the user. Command-line utilities are more suitable for scripting and server maintenance.

4.1 Kernel messages

To receive messages from the kernel in userspace, there has to be some facility to pass such messages. Kernel messages are not always warnings, errors or debug messages. When configured, the kernel can pass statistics or other monitoring information to be logged afterwards.

The simplest way for the kernel to report some data is via `printk()`, and then the user can access the kernel message buffer by `dmesg` command (the binary connection is done via `/proc/kmsg`) or by syslog daemon. Most of the data is generated in bootup stage, so it can be used to check the configuration of kernel and hardware (like duplex NIC etc.).

4.1.1 debugfs

The syslog daemon is useful for logging all the data from all processes in a computer, but is not very practical for transferring huge amounts of data from the kernel. Special filesystem, `debugfs`, can be used instead.

`debugfs` is a virtual filesystem developed to export debugging information to the userspace, highly used by kernel developers. Creating a file in `/proc` requires too complex kernel programming, files in `/sys` can mostly contain only one value and not sequential data. The best way to put lots of debugging information to userspace is by using special filesystem and `debugfs` has been created by Greg Kroah-Hartman to become a standard debugging filesystem.

Programming the debugging filesystem is a simple task for kernel developer, to export values, you only have to create a directory and then a file filled with sequential data or you can use predefined functions to export only one variable into a file. Exported variables are also writable which makes the filesystem ideal for debugging purposes.

4.2 Sysfs

The linux kernel provides a special filesystem **sysfs** to access device information from userspace (like **procfs** provides kernel parameters) and it is usually mounted at **/sys**. The goal of the **sysfs** filesystem is to expose the hierarchical relationships among all components of the device driver model.

The related top-level directories of this filesystem are:

- **block** – block devices, independently connected to the bus
- **bus** – buses in the system with all connected devices
- **class** – types of devices in the system
- **devices** – devices recognized by kernel, organized per bus
- **firmware** – files to handle the firmware of certain devices
- **module** – information generated by (parameters of) loaded modules
- **power** – files to handle the power state of certain devices

4.3 Networking tools

Tools for monitoring network activity are primarily focused on the throughput of a NIC. Basic information is obtained from hardware device counters in the form of byte and packet count (received, sent or aggregated), which can then be transformed into bytes per seconds (this information can be found in **/proc/net/***).

4.3.1 ifconfig

Essential tool for configuring network interfaces. Aside from the configuration functionality, it also reports many properties and statistics of the interface. The total sum of transferred packets, dropped and erroneous packets since bootup time is shown for requested interfaces.

4.3.2 iftop

Simple, yet well arranged command line tool for monitoring the current bandwidth usage on network interfaces. This tool also provides filter for packets to be counted in the statistics.

4.3.3 ifstat

A command-line tool for interface statistics reporting. The output (see A.1) is similar to **iostat** or **vmstat**. Without any parameters given, it displays the current transfer rate (in and out) for all interfaces that are up. The output is written after a polling interval (default 0.1s, can be changed) infinitely (the number of polls can also be specified).

When the parameter **-i** is set, only selected interfaces will be displayed. Using parameter **-T** a total transfer rate is printed. To get the timestamp for current values, add a **-t** parameter. The application uses multiple sources (or drivers) to receive the data, like **proc**, **SNMP** and others, see **man ifstat** for all definitions.

4.3.4 dstat

Dstat is a versatile tool combining **iostat**, **vmstat** and **ifstat** to show multiple statistics in its output. It also uses colorized output to show the magnitude of printed values. For example network usage can be seen in a context with disk usage. The decolorized output is shown in appendix A.2.

4.3.5 netstat

Netstat can printout much statistical information about the network interface or underlying network connections based on the type of information requested. See A.3 for outputs based on different settings.

- **netstat** – Lists open sockets of all families (**-A unix,ip,...** selects the family of sockets) and related processes (parameter **-p**).
- **netstat -M** – IP masquerading statistics (if enabled in the kernel).
- **netstat -r** – Kernel routing table, equals to the output of **route**.
- **netstat -g** – Displays multicast group membership for all interfaces.
- **netstat -i** – Kernel interface table, similar to **cat /proc/net/dev**.
- **netstat -s** – Overall statistics for all network protocols. There are many statistical values (e.g. TCP retransmission count, ICMP message count, IP packets forwarded, etc.).

4.3.6 lsof

This command lists open files in the system. Parameter **lsof -i addr** lists internet connections to/from given address (if no address specified, all connections are listed). Where the **netstat -A ip -p** shows only open connections, **lsof -i** lists also listening sockets and UDP connections (A.4).

lsof also supports filtering based on user or process name, SELinux policies and more. For complete usage description see **man lsof**.

4.3.7 nfsstat

Nfsstat is a monitoring tool for NFS. Prints out statistics about client and server RPC calls and transactions (all versions by default, override by **-[234]**). With the option **-m** it lists all mounted volumes via NFS. Example output can be found in appendix A.5.

4.4 Disk tools

In the case of disk monitoring, not only the throughput is important, but also the response time of the disk (i.e. queue status and seeking time). As for the space usage, it is not the goal of this work to monitor the disk space usage (e.g. via `df` or `du`).

4.4.1 `smartctl`

This tool is a part of `smartmontools` and it is used to monitor disk's SMART information and executing a self-test. Use this tool to monitor the hardware status of the disk.

Disk hardware information can be obtained via `smartctl -i` (device identification and information) or `smartctl -a` (for SMART related values and errors). Other parameters are used to set the SMART capabilities, run/abort testing, enable features, etc. Full parameter list and description can be found in `man smartctl`.

4.4.2 `lsof`

The `lsof` command can be widely used for other files than only network sockets. The filtering capabilities are very useful when looking for a process or a user/group (by `-p` or `-u/-g`) that has an extensive amount of opened files. It can also list only files opened in given directory (and subdirectories) when `+d (+D)` parameter is used.

See documentation in `man lsof` and example outputs in A.4 for usage information.

4.4.3 `iostat`

`Iostat` reports CPU statistics and I/O statistics for disks and partitions. The disk utilization statistics include transfers per second rate, block read/write statistics and data transfer rates. More specific information (when parameter `-x` is given) includes read/write requests merged to queue/sent to device/completed per second, average service times for the device and device data transfer speeds.

`Iostat` also supports NFS statistics when `-n` is specified (only for kernel 2.6.17 or later). See appendix A.7 for examples.

4.4.4 `dstat`

As mentioned in previous section, the `dstat` command provides complex and centralized statistics in one combined output.

4.4.5 `filefrag`

`Filefrag` is a reporting tool for ext2/ext3 filesystems, that can be used for other filesystems as well. It reports how much a file is fragmented (on how many series of blocks it resides). It is best for a file to be in one consecutive chunk instead of being distributed over the disk, because more seeking is needed to read all the data of a file.

The problem of this tool is that it only scans one regular file at a time and the output is text-only, which would not really useful for larger scale. This problem is addressed later in this work.

4.5 Kprobes

Kprobes, a new feature in the Linux 2.6 kernel, allows for dynamic, in-memory kernel instrumentation. To use kprobes, the developer creates a loadable kernel module which calls the kprobes interface. These calls specify a kernel instruction address, *probe point*, and an analysis routine, *probe handler*.

Kprobes arrange for control flow to be intercepted by patching the probe point in memory (adding a breakpoint before given instruction), with control passed to the probe handler. Kprobes has been carefully designed to allow safe insertion and removal of probes and to allow instrumentation of almost any kernel routine. It lets developers add debugging code into a running kernel. Because the instrumentation is dynamic, there is no performance penalty when probes are not used.

The basic control flow interception facility of kprobes has been enhanced with a number of additional facilities. *Jprobes* make it easy to trace function calls and examine function call parameters. *Kretprobes* are used to intercept function returns and examine return values. Although it is a powerful system for dynamic instrumentation, a number of limitations prevent kprobes from a broader use:

- Kprobes do very little safety checking of probe parameters, making it easy to crash a system through accidental misuse.
- Safe use of kprobes often requires a detailed knowledge of the code path to be instrumented.
- Due to references to kernel addresses and specific kernel symbols, the portability and reusability of the instrumentation code using kprobes interface is poor.
- Kprobes do not provide a convenient mechanism to access a function's local variables, except for a Jprobe's access to the arguments passed into the function.
- Although using kprobes doesn't require a kernel build-install-reboot, it does require knowledge to build a kernel module and lacks the support library routines for common tasks.

These problems create a significant barrier for potential users. A script-based system that provides the support for common operations and hides the details of building and loading a kernel module will serve a much larger community, which is one of the main motivations for creating SystemTap.

4.6 SystemTap

The goal of SystemTap is to provide an infrastructure to simplify the gathering of information about the running Linux kernel, so that it can be further analyzed. This can assist in identifying the underlying cause of a performance or functional problem. The recent addition of Kprobes to the Linux kernel provide the needed support but does not provide an easy-to-use infrastructure. SystemTap provides a simple command-line interface and scripting language for writing kernel instrumentation scripts.

The essential idea behind a systemtap script is to name events and to give them handlers. Whenever a specified event occurs, the Linux kernel runs the handler as if it were a quick subroutine, then resumes. There are several kind of events, such as entering or exiting a

function, timer expiration, or the entire systemtap session starting or stopping. A handler is a series of script language statements that specify the work to be done whenever given event occurs. This work normally includes extracting data from the event context, storing them into internal variables and printing results.

Using SystemTap is simpler, much safer and easier to use than directly using Kprobe modules. As systemtap scripts can be found in the community, reusability becomes easy. One of the main goals of SystemTap is enabling a production-environment use, which means having a crash-proof tool at hand.

4.6.1 Implementation

SystemTap takes a compiler-oriented approach to generating instrumentation. Architectural overview can be seen in figure 4.1.

First, SystemTap parses user's script (**probe.stp**) and resolves library functions, filling in needed data from current kernel debugging information. Based on such created parse tree, a C-source code is generated (**probe.c**).

The **probe.c** file is compiled into a regular kernel module (**probe.ko**) using the GCC compiler. The compilation may pull in support code from the runtime libraries. After GCC has generated the module, the SystemTap daemon is started to collect the output from this instrumentation module.

The instrumentation module is loaded into the kernel, and data collection is started. Data from the instrumentation module is transferred to userspace via **relayfs** (data relaying filesystem, see [18]) and displayed by the daemon. When the user hits Control-C the daemon unloads the module, which also shuts down the data-collecting process.

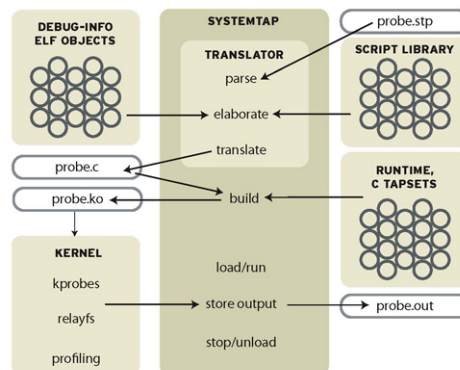


Figure 4.1: Flow of data in SystemTap (from [17])

4.6.2 Installing

SystemTap makes heavy use of compiler debugging information in the kernel binaries. Therefore to use systemtap scripts, kernel has to be compiled with such debugging information.

There are few possible ways to achieve this, varying from distribution to distribution. Some distributions provide a special kernel package (Debian – **linux-image-debug**, RHEL – **kernel-debuginfo**) that can be precompiled. Another way is to recompile the kernel manually with **CONFIG_DEBUG_KERNEL=y** in the kernel config file.

The systemtap daemon and command-line interface can be installed using the **stap** or **systemtap** package (see your distribution package tree or visit the systemtap homepage – [15]).

On some distributions there is a glitch in compatibility. SystemTap tries to open uncompressed kernel image using a path `/lib/modules/‘uname-r’/vmlinux`. When systemtap complains about missing debugging symbols, check the existence of the image in that path and when missing, simply create a symlink to it (e.g.

```
ln -s /usr/src/‘uname-r’/vmlinux /lib/modules/‘uname-r’/vmlinux).
```

4.6.3 Tracing using SystemTap

The systemtap input consists of a script, written in a simple language. This language describes an association of handler subroutines with probe points. Probe points are abstract names given to identify a particular place in kernel/user code, or a particular event (timers, counters) that may occur at any time.

Handlers are subroutines written in the script language, which are run whenever the probe points are hit. Probe points correspond to **gdb** breakpoints, and handlers to their command lists.

The language resembles C, itself inspired by the old UNIX tool **awk**. The language is lacking types, declarations, but adding associative arrays and simplified string processing. The language includes some extensions to interoperate with the target software being instrumented, in order to refer to its data and program state. Complete language reference can be found in Documentation section on the SystemTap homepage [15].

A simple example script **kprobeio.stp** (see appendix B.1) uses one global variable `count_generic_make_request` and three probes. The probe locations are: **begin**, which executes before any other probe in the script and logs the start of probing; **end**, which executes on the instrumentation shutdown; `kernel.function("generic_make_request")`, which is called due to a breakpoint inserted into the kernel at the point where the actual `generic_make_request()` function is called.

Running the script is very simple, just typing **stap kprobeio.stp** into the command-line is sufficient. The user has to have root privileges to insert probes into a running kernel of course. Output of this simple script is following:

```
starting probe
ending probe
generic_make_request() called 24 times
```

4.6.4 Tapsets

Tapsets are similar to *libraries* known from modular programming. There are two main types – scripted tapsets and C-language tapsets. The scripted tapsets are common for defining aliases (`kernel.syscall.read` → `sys_read`), creating small but widely-used functions (e.g. formatting output) or some helpful commands to parse data into a structure, etc. C-language tapsets are mostly used by developers for the consistency of this programming language and to create more sophisticated and optimized functions.

4.6.5 Safety

SystemTap is designed for safe use in production systems. One implication is that it should be extremely difficult, if not impossible, to disable or crash a system through use or misuse of SystemTap. Problems like infinite loops, division by zero, and illegal memory references should lead to a graceful failure of a SystemTap script without otherwise disrupting the monitored system.

The developers avoid privileged and illegal kernel instructions by excluding constructs in the script language for inlined assembler, and by using compiler options used for building kernel modules. SystemTap incorporates several additional design features that enhance safety. Explicit dynamic memory allocation by scripts is not allowed, and dynamic memory allocation at runtime is avoided.

4.7 blktrace

Tracing block device's I/O actions is another new addition to Linux kernel. Using **debugfs** it sends per-I/O-action data to the userspace. This is the main difference between **blktrace** and **iostat**. **iostat** can report only global statistics, but **blktrace** has the capability of tracing each and every I/O operation requested from the kernel.

This tracing has a very low overhead (about 2% of application performance according to developers). The kernel module reports *events* that happen in the block layer (queue insertion, sleeps, merges, start and completion of I/O request, ...) through the **debugfs** and the monitoring side reads it from there. However **blktrace** is not an analysis tool, it is only the mean of transporting needed information through **debugfs** into userspace.

Events extracted by **blktrace** are in binary form and can be stored or piped to the extraction utility called **blkparse**. This utility can generate statistics from given binary data. The output data contains: device ID, CPU ID, sequence number, timestamp, PID, event type (queue, request, complete, ...), block address, block size and process name.

When the tracing is finished, **blktrace** outputs a summary containing the average throughput, number of merges, queued requests, completed events, etc.

The last tool using traced statistics is **btt**, which shows the lifetime information about an I/O request. The lifetime information traces the processing time of a request and shows all the parts of the request-path (before insertion to the queue, being idle and time being active on the device).

To enable block tracing support, set **CONFIG_BLK_DEV_IO_TRACE=y** in the kernel config file and install the

blktrace application package. Before using the tool itself, **debugfs** has to be mounted and the path given to **blktrace** (default value is **/sys/kernel/debug**).

Example outputs of all programs mentioned above are in appendix A.8.

4.7.1 Events

By default, **blktrace** collects all events that can be traced. To limit the events being captured, you can specify one or more filter masks via the **-a** option. The event types and possible mask values (they are not disjunctive) are following:

- **barrier** – *barrier attribute*

Some requests can have this flag set. Before a barrier request is started, all preceding requests in the queue must be finished and all following requests can be started only

after this request has been completed. This creates a firm request ordering and flushing to disk. It is used mostly for journalling.

- **complete** – *completed by driver*
Event marking an end of the I/O operation. Useful for time measurement.
- **fs** – *filesystem request*
These are all read/write operations on an IDE disk (having specific disk location and size). Useful for data tracking.
- **issue** – *issued to driver*
Event occurring when the request is taken from scheduler queue and handed over to the hardware driver by the kernel. Very useful for measuring seek times and disk hardware attributes.
- **pc** – *packet command event*
There are all SCSI commands with command data block. Useful for monitoring SCSI driver events.
- **queue** – *queue operation*
All operations done on the scheduler queue (addition, merge, etc.). Useful for scheduler profiling and average queue-time monitoring.
- **read** – *read trace*
Only read operations (transferring data from the disk).
- **requeue** – *requeue operation*
The scheduler can remove a request from the queue and requeue it afterwards to increase performance.
- **sync** – *synchronous attribute*
This flag is only set for request that are synchronous (like writing into files with `O_DIRECT` flag).
- **write** – *write trace*
Only write operations (transferring data to the disk).

Synchronize and barrier flags are set on the beginning (if needed or specified). An example datapath for reading some data from a disk (and emitting appropriate flags) is visualized on figure 4.2 – all events would be flagged **read** because we are reading data.

The **blkparse** utility (or **btrace** script that combines **blktrace** and **blkparse**) shows the type of events captured. One of the output columns can contain RWBS symbols to show whether the request was a read or write request and whether it has been called with a barrier or synchronize flag.

The second flag column can contain following symbols – **ABCDGIMPQSTUX**, the meanings are:

- **A** – *I/O was remapped to a different device*
For stacked devices, incoming request is remapped to device below it in the I/O stack. The remap action details what exactly is being remapped to what.

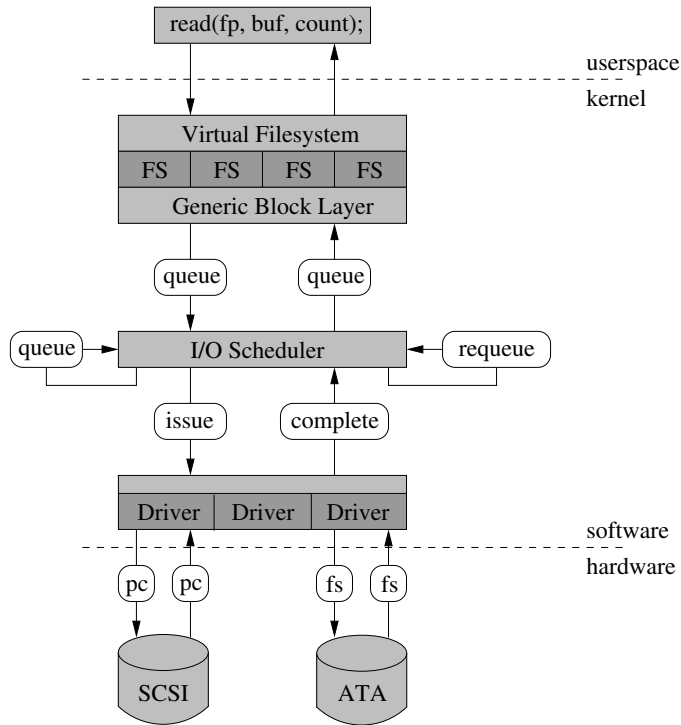


Figure 4.2: Datapath and emitted flags for blktrace

- **B** – *I/O bounced*

The data pages attached to this block-io structure are not reachable by the hardware and must be bounced to a lower memory location. This causes a big slowdown in I/O performance, since the data must be copied to/from kernel buffers.

- **C** – *I/O completion*

A previously issued request has been completed. The output will detail the sector and size of that request, as well as the success or failure of it.

- **D** – *I/O issued to driver*

A request that previously resided in the I/O scheduler queue has been sent to the driver.

- **F** – *I/O front merged with request on queue*

Same as the back merge, except this request ends where a previously inserted request starts.

- **G** – *Get request*

To send any type of request to the scheduler, a memory structure container must be allocated first.

- **I** – *I/O inserted into a request queue*

A request is being sent to the I/O scheduler for addition to the internal queue. The request is fully formed at this time.

- **M** – *I/O back merged with request on queue*
A previously inserted request exists that ends on the boundary of where this request begins, so the I/O scheduler can merge them together.
- **P** – *Plug request*
When a request is queued to a previously empty block device queue, scheduler will plug the queue in anticipation of future requests being added before this data is needed.
- **Q** – *I/O handled by request queue code*
A request is being formed to be sent to scheduler.
- **S** – *Sleep request*
No available request structures were available, so the issuer has to wait for one to be freed.
- **T** – *Unplug due to timeout*
If no more requests are sent to the plugged queue, scheduler will automatically unplug it after a defined period has passed.
- **U** – *Unplug request*
Some requests are already queued in the device, start sending requests to the driver. This may happen automatically if a timeout period has passed (see T) or if a number of requests have been added to the queue.
- **X** – *Split*
On RAID or device mapper setups, an incoming request may straddle a device or internal zone and needs to be chopped up into smaller pieces for service.

For a detailed description see the blktrace and blkparse userguide [13].

4.7.2 Output storing

Although the output data of blktrace is of a huge quantity, it should be stored for later analysis. But storing such large data output comes with many difficulties. There are few ways to achieve the backup of blktrace outputs:

- **Physical disk** – enough space, easy to use, but system calls are used for writing (performance impact) and the I/O subsystem that should be monitored is part of the test-harness now
- **RAM disk** – although little impact on CPU and I/O, limited in space and fixed memory allocation
- **tmpfs** – only utilizes RAM when needed, less I/O intensive, but still some impact, RAM drain can be large, limited size

Chapter 5

Solving high I/O workload situations

Running a server in an enterprise environment comes with a great responsibility at many times. Most of such servers have to be available all the time, where available doesn't only mean that the server is running, but that it is also responsive and stable.

To meet these requirements, administrators must have adequate tools for performance monitoring and analysis, which are not only used for long-term monitoring and thus optimization, but also for dealing with problems in the shortest time possible.

This particular thesis is focused on such tools, particularly on I/O oriented tools. The reason is simple: determining a bottleneck or an error bound to a CPU is a well-mapped task, one only has to run the `top` command to see which processes are CPU-intensive and based on that information limit them or increase the CPU power. The same situation applies for memory-oriented deficiencies.

Disk operations on the other hand are more complicated. There are many layers on the datapath and many components with different interfaces, data rates etc. That is the main reason for not having a complex set of tools and methods for dealing with disk I/O inefficiency. Following chapter tries to provide such solution.

Next sections of this chapter describe a method to be applied by the administrator to deal with I/O oriented deficiencies on a running server (e.g. fileserver, webserver, etc.). I will first show needed steps to be taken to determine the source of the slowdown from the very beginning. Using these methods I will present some common real-life situations and use previously defined steps to deal with them.

5.1 Detecting the global source

The first step to be taken is to determine the basic scope of the slowdown. There are four main areas to be focused on – CPU load, memory allocation, network bandwidth usage and disk throughput/responsiveness.

5.1.1 CPU load in userspace

The first thing that people think of when a computer responds slowly is that the CPU load is too high. This is not really true, most of time the real problem lies in disk response time, because operating systems (particularly on desktop computers) can easily deal with CPU-intensive processes and still preserve overall responsiveness.

Nevertheless, a high CPU load can easily slow down the whole server. To detect high CPU load one can use many tools from `top` to graphical applications. The source of CPU load information is the `/proc/stat` file, which can easily be parsed in a script or binary application. First few lines of the output contain data bound to each processor of the machine (the first line aggregates all other CPU lines):

```
cpu 82419 351399 113941 4983746 57796 11335 39058 0
cpu0 35202 182639 56921 2482112 37595 5361 20027 0
cpu1 47217 168759 57019 2501633 20201 5973 19030 0
```

These numbers identify the amount of time the CPU has spent performing different kinds of work. Time units are hundredths of a second. The meaning of shown columns is as follows, from left to right: user, nice, system, idle, iowait, irq, softirq. For more information see [20].

The percentual load can be computed from the difference of these numbers in given time slot and computing the fraction part of each type of work.

5.1.2 CPU load in kernel

High kernel CPU load can also be detected by previously mentioned tools (usually labeled as `sys` load). The problem lies in determining the exact source of such load and also in resolving the problem.

Some processes can use system calls too much (`fork()` and `exec()` for example), which is detected by higher system time of a process. The only resolution is changing its behavior by rewriting the program's source code.

The second (and worse) possible case of high system CPU load is more difficult to examine. Kernel manages all the low-level operations needed for the computer to work, process and memory management, I/O, etc. Kernel should utilize its jobs to be executed only in relatively small batches and when the CPU is more idle, but sometimes there are jobs that have to be executed immediately.

These situations are more common when a module has a bug or a built-in driver loops in the kernel. Finding the real source in the kernel is highly difficult and is mostly resolved by kernel or driver developers.

5.1.3 Memory allocation in userspace

Memory allocation as such is not really a source of a slowdown. When a process cannot allocate more memory it usually dies, so the problem is not a slow responsiveness but wrong functionality. However, this also has to be detected and resolved.

Due to the existence of virtual memory, space itself is usually not a problem, but swapping is. When a memory segment is swapped, it has to be written to disk and read from it afterwards. This can cause severe amounts of disk usage thus creating a high I/O workload and slowing down the computer. The extreme case is called **thrashing**, a situation where the most time is spent on swapping instead on actual work.

Detection of high memory allocation is also relatively simple. A command line utility `free` provides the essential data about memory usage.

	total	used	free	shared	buffers	cached
Mem:	3105548	2875064	230484	0	17740	2422076
-/+ buffers/cache:		435248	2670300			
Swap:	4008176	260	4007916			

The first line's memory usage (first three columns) is not really useful, because linux kernel doesn't free memory immediately (it stays cached), but only when needed. The **buffers** column is the amount of memory used for directory cache and other kernel buffers. Special type of such buffers is the filesystem buffer cache (separated column **cached**), that holds all the data read/written from/to a disk for further use. This cache grows and shrinks on demand, it is only freed when the memory is needed for running processes.

The second line shows the "real" memory usage (without cached memory and buffers). This is the addition in the new format of this tool. Last line shows the amount of memory swapped to disk. When the swap space is used, disk I/O starts to be a factor, so the higher the swap, the most likely a slowdown occurs.

It has to be said, that having the swap space used (although it may be of nearly 100%) doesn't always mean that thrashing will occur. There may be much data swapped and not used for a long time. The crucial factor is the swap partition usage. To determine read/write transfer speed, **vmstat** tool can be used. The output is following:

```
procs  -----memory-----  --swap--  ---io---  -system--  -----cpu-----
 r  b  swpd   free   buff  cache   si   so   bi   bo   in   cs  us  sy  id  wa
 0  0   260 121976  35556 2487904   0    0  561  134  497  663  3  2  95  1
```

The **--swap--** column shows written (**so**) and read (**si**) data in bytes from previous measuring (or average from bootup, when delay parameter not given, see **man vmstat**).

Kernel reports the memory information via **/proc/meminfo** (which is also where **free** looks) and it is the first place to use when writing scripts for automatic memory monitoring. The file output is self-explanatory.

5.1.4 Memory allocation in kernel

To see the kernel memory usage statistics, see **/proc/slabinfo** that reports the kernel *slab cache* usage. This cache is used for objects having the same size, thus saving memory. Too many allocated entities can cause a problem in memory allocation. A command-line tool similar to **top** is focused on slab memory allocator – **slabtop**.

Another important allocator is the *buddy allocator*. It is used to minimize memory fragmentation created by memory allocation. Kernel exports statistics in **/proc/buddyinfo**.

Kernel itself allocates memory mainly through slab caches and has some statical amount of memory allocated all the time. More memory is of course allocated when needed. The largest memory-consumers in kernel are buffers. Buffers are used for transferring data into userspace and in many cases (disk, network, etc.), such buffers are allocated the whole time, thus decreasing the amount of useable memory for processes.

5.1.5 Network bandwidth usage

When talking about servers, the network connectivity has to be taken in account too. Due to the fairness algorithm of TCP protocol, a connection shares the bandwidth equally with all other established connections. When there are too many connections transferring data, the responsiveness can be low.

The solution for this problem is easy to provide, but hard to make. Lowering down the number of possible connections can increase transfer speed, but also decreases the number of transactions at one time. Limiting a connection speed by quotas or daemon configuration is another way of achieving balance, but can be counterproductive when only few connections are established.

Monitoring network bandwidth on an interface is easy through a command-line graphical tool `iftop`. The raw data can be obtained through `ifconfig` where the packet count (measured from bootup) is shown or directly from kernel in `/proc/net/dev`.

5.1.6 Network latency

Currently, the network usage is usually not a problem. Network cards are capable of really high speeds and configuring quotas to ensure balanced throughput is trivial. Network latency however can still be a problem. Besides the "outside" latency (latency on the network path – switches, long distances, LAN firewalls,...), latency can occur on the server itself.

As for TCP, bad setup of maximum read/write buffer (both have to be set, because TCP chooses the smaller one) that are used to compute the *congestion window*, can cause major latency because the TCP fairness algorithm narrows the window down. More on setting network buffers in 3.2.2.

Another latencies can occur when the packet processing time becomes too long. Packets are processed by the kernel when iptables are compiled in. Various other firewall tools (mainly packet inspection tools, IDS/IPS) prolong the packet processing time by checking the contents of data payload.

5.1.7 Disk I/O

Finally, when other possible sources are eliminated, disk I/O subsystem should be analyzed. The analysis is not that simple as in other areas covered before. There are many reasons why the disk can be responding slowly, as there are many ways processes can read/write data.

When a process reads large amounts of data, the disk maximum transfer rate and bus bandwidth are used. A disk has a limited transfer rate (ranging greatly depending on its technology, rotational speed, etc.) that can be determined by `hdparm -t`.

The disk maximum transfer rate is nowadays much slower than the bus bandwidth, but the bus is usually shared by more disks, so the bus bandwidth limit can be reached easily on a server. For SATA it is 150 or 300 MiB/s, for SCSI 320 MiB/s, for SAS 375 MiB/s. SATA disks are usually connected as single drives to a single bus, but using an expander (device connecting more disks via one bus), the bus can be easily choked although disks are still running below the maximal bandwidth capacity. The same goes for external SATA/SCSI devices.

Detecting a high disk usage is the simpler task in disk monitoring. Again there are some tools that can be used, like `iostat` or `dstat`, which show current transfer rates for disks and partitions. The kernel data can be obtained through `/proc/diskstats` (the output format is very similar to `/proc/stat`).

Other way, the disk can become slow in the view of waiting processes, is a long data retrieval time. Long data retrieval time can be caused by many factors – high or slow disk seeking, disk errors, too many requests in the waiting queue,... . Determining the exact cause is a non-trivial task and it is much more complex than a high-bandwidth detection

mentioned in the previous paragraph. Some information is given by the `iowait` value for CPU load, but it is an aggregated value for all I/O subsystems.

In both cases, the I/O waiting time grows and the processes that use blocking reads or writes become unresponsive. It is then the goal of the administrator to determine the exact source of such problems and to deal with them. Finding the problem-causing processes/files/disks is the focus of the next section.

5.2 Focusing on the disk I/O

After determining the global source of system slowdown, some actions can be taken immediately (stopping CPU intensive processes, configuring daemons, upgrading hardware), but when the source is selected to be the disk I/O subsystem, there are too little information to make action at this point.

To properly analyze the problem, more exact information has to be gained, preferably which process and which file(s) are involved. Following steps can help determine more precise source of a system slowdown.

5.2.1 Disk and partition

The first information to be determined is the disk and partition where the problem resides. This information is itself very valuable. When only one disk of the server is heavily used during a longer time period, files could be split among multiple disks to prevent high disk usage and a risk of failure. Also filesystem change on a partition can be reconsidered.

Of course such steps cannot always be taken in practice. Distributing data among several disks is possible for file servers or data storage servers, but not really for database servers, where the data has to be placed in given directories.

The affected disk and partition is the main parameter passed to more sophisticated tools (as in following scenarios, a disk device is passed to `blktrace`). `iostat`, `dstat` or reading direct data from `/proc/diskstats` should be used to obtain the particular partition affected.

When the transfer rate for no partition is of a higher value, the problem can reside in the data retrieval time (errors on disk, long seeking, etc.) which is harder to detect and can be a reason for monitoring each device separately.

5.3 SystemTap scripts for blktrace

Having known the disk to be monitored, `blktrace` is the right tool to analyze all I/O transactions executed on that disk. Such data then lead to understanding where the problem lies (long waiting time, disk seeking, etc.).

However, the `blktrace` subsystem provides big amounts of data gathered by the tracing process. The tool itself is very powerful and sufficient for the goal of this work. On the other hand the data is not easy to follow and parse in sufficient time, so formatting and parsing the output is the key element for providing simple, yet efficient method for the administrator to use on the server.

Parsing outputs from `blktrace` would be very intensive for the server and results would not be ready in sufficiently short time. Combining `blktrace` data with SystemTap scripts gives us relatively easy, sufficiently portable and reliable way to create realtime results and statistics from `blktrace` data, because the data are taken directly from function calls inside

the Linux kernel. The SystemTap developers and community gather scripts for many uses and different kernel modules and then shares them for public use, so updating, maintaining and support (through mailing lists mostly) is at a satisfactory level.

One particular article on the systemtap mailing list from March 2007 (available at [19]) is focused on SystemTap and blktrace integration based on the function call `__blk_add_trace()`. Tom Zanussi created a tapset and some scripts for basic I/O statistics. Those scripts can easily be used to provide the essential information about the I/O workload and to help the administrator to achieve the goal mentioned in this thesis.

The script and tapset package are available at the mailinglist website ¹. I will shortly summarize those scripts for further use.

5.3.1 Parameters

SystemTap scripts do not take parameters themselves (except for a filename, changes can be done quickly in the code when needed), but they gather data from `__blk_add_trace()` function call, which means, that they only produce outputs, when `blktrace` is running. Parameters for `blktrace` command-line utility were discussed earlier, in short, you can focus only on reading/writing, synchronous events, or only on device or queue operations. The main parameter being the traced device.

Running `stap` using these scripts and tapsets is simple: `stap -I tapsets <name>.stp`.

5.3.2 countall.stp

This script just keeps a running count of all `blktrace` events that have occurred over the run, broken down by event type and read/write direction. The output from a kernel compile can be found in appendix B.2.

The script output gives an overview of I/O operations happening during selected period of time, useful for tuning and optimization of the system.

5.3.3 spectest.stp

This script provides *speculative tracing*, it keeps the most recent I/O requests in memory and when a condition is met, it stops tracing, enabling the user to check the last few requests for possible problems. The condition implemented in the script is checking the `q2c` time (the time between the request entering a device queue and the moment data successfully returns to userspace) for not going over one second (the time period can be easily changed in the script's code).

Using this script, one can detect long seeks that prolong the device's `d2c` time (time between issuing the request to the device and data returning to userspace) and too many requests waiting in the queue (usually from too many processes using the device at one time) – the `q2d` time. When needed, changing the condition in the script from `q2c` to `d2c` or `q2d` conditions, more precise tracing can be done. Although it is not really necessary.

Using the sector number in output (see B.3), one can locate the data that caused `spectest` to stop. When multiple files are seen having very different sector numbers, the problem mostly lies in the data physical position on the disk, thus increasing seeking time. The full output of `blktrace` tool should be saved somewhere to be analyzed later. In this case the sector number gives the exact point in the output to be looked at.

¹<http://sourceware.org/ml/systemtap/2007-q1/msg00485/blktrap.0.tar.gz> (last visit 04.02.2008)

5.3.4 iotop.stp

This is the most "user-friendly" script providing similar output as the well-known `top` tool. It periodically (5 seconds) displays top 20 I/O producers in the system. In the output (see B.4 – cold starting firefox) there is the PID, process name and I/O statistics for transfer rates (total, read and write). Don't forget, that the values are measured in 5s intervals.

Such tool is highly valuable to the administrator when a high workload occurs and it lets him find the right process to deal with. When the highest producer of I/O workload is found, it is on the administrator to choose the next action (stop, kill the process or setup the process properly). If more data is needed, using the process name and `topfile.stp` script, the exact file (or files) the process reads can be detected.

Besides the output formatting, the main benefit of this script is the process-to-request connection. Because most of the write requests are deferred ones, the actual write is issued by `pdflush`. The script tracks dirty pages to determine what process is responsible for the write request by retrieving the dirtying PID associated with the page in `bio` structure in kernel.

Running `iotop.stp` requires extra parameters to increase memory limits, because SystemTap default limits are set too low (will be changed in future versions):
`-DMAXMAPENTRIES=10000 -DMAXACTION=10000`.

5.3.5 topfile.stp

Sometimes it is useful for the server workload analysis to know which files are frequently read/written from/to, either in the global scope or only for one process that is examined (see previous section). The output of this script does exactly the job, during its runtime it gathers `blktrace` data and after shutdown it prints the I/O statistics for all files used by the process(es). The truncated example output for the firefox process can be found in appendix B.5.

To set the process to be traced (instead of the global scope), uncomment the `watchme` global variable in the script and change it appropriately. This script also requires additional parameters to increase limits: `-DMAXMAPENTRIES=10000 -DMAXACTION=10000`.

5.3.6 traceread.stp

This script gives a detailed sequence of events with their timings and time-delays from entry into `vfs_read()` function to this function return. Tracing can be done for all files or only one specified as a parameter (give `all` for all files or a filename). The output shows where the longest delay is, helping the administrator to find bottlenecks. The truncated example output can be found in appendix B.6.

5.4 Summary

Choosing the right solution varies from problem to problem (based on the reason of it) and it is on the administrator to do such thing. Some ideas were mentioned in previous chapters and optimization techniques were discussed in chapter 3.

To summarize, I present the main categories of solutions to most of the problems, ordered by severity of changes:

- **Proper configuration** – consulting manual and setting basic options for daemons:
See documentation and manuals for appropriate software.
- **System optimization** – configuring buffers, virtual memory, filesystem, etc.:
See chapter 3 for more information on GNU/Linux optimization.
- **Load balancing** – moving files to separate disks and bus channels avoids bottlenecks:
Depending on the situation, not always possible.
- **Access policy** – reducing processes/people accessing data at one time:
Mostly application-side setting, disk/CPU quotas.
- **Hardware upgrade** – should be used only when optimization failed, sometimes unavoidable

Chapter 6

Scenarios

To provide practical usage examples of previously presented tools and methods, this chapter gives some of the most common situations related to I/O slowdown on servers and/or desktop systems. Each section describes a situation and known information as a prerequisite for uncovering the source of the problem and giving a solution to it.

Going from simpler situations to the ones more complex, this chapter creates a basic "guide" for I/O-oriented slowdown resolution. First few situations do not cover the area of I/O subsystem, but show solutions for uncovering the problem in other scopes. Use these scenarios as a guide to eliminate other areas of interest before stepping into I/O layer.

Network oriented scenarios given further apply mostly to situations, when the slowdown is reported from the outside (remote client), because the response on the client side can be slow, but local users do not usually see any problem.

6.1 Non-disk scenarios

6.1.1 High CPU load

All processes have a slow response. The important symptom is that it affects all types of processes, not only disk-oriented (web and file servers, copying files, ...). Even if this information was not given, the detection is still simple, just by running `top` the problematic process is discovered.

Having a high system CPU load can be a sign of a slowdown due to too many system calls, but it doesn't have to be so. It is normal for file servers (or any servers dealing mainly with I/O), that the kernel is called more often thus creating a higher system CPU load. Such servers are not optimized for running any other applications, so the higher latency for userspace applications is not really a problem.

6.1.2 High network usage

Connection speed (read and/or write, depending on the situation) is low. This can be seen only when transferring larger amounts of data, connectivity itself stays at a good level. To test whether the problem lies in the network or in data retrieving from the server itself, network interface statistics have to be reviewed.

The `ifstat` tool displays bandwidth usage for given interfaces. If the total bandwidth on an interface is reaching the maximum for the given line (and of course interface card), the problem has been found. To track-down the remote client(s), `iftop` can be used, because

it displays bandwidth related to local and remote address and port. Network statistics are more precise for TCP than UDP streams, because TCP is a connection-oriented protocol and relevant connection is established for a longer time period, enabling better information gathering for statistics.

There is yet one situation caused by a high network usage. When there are more interfaces (or a gigabit network card), the total bandwidth of the network subsystem can be higher than the maximal transfer rate of a disk. In that case, it is important to know the disk partitioning of the server. If there is for example only one disk on the server and the combined transfer rate of all network cards gives the maximum transfer rate for this single disk, the problem is not in the system configuration nor the disk, the server is just fully utilized. If the problem is not the high network load, more data gathering is needed.

To gather more data in this case, it is not necessary to use sophisticated tools, but instead study the logs or monitoring tools of communicating daemons (ftpd, sshd, apache). The data should be sufficient to locate proper user and file(s) involved in the slowdown.

6.1.3 Too many network connections

This scenario is also a well-known case of DoS attack. Creating many connections on a remote server can deplete the number of free ports for the transport protocol (state-oriented TCP) or create too many processes when the server-side daemon uses `fork()` for every incoming connection. Servicing too many clients at once can also be a problem for some other server applications.

Interface bandwidth statistics are useless here, monitoring of open sockets (files) and connections is needed. The `netstat -tu` command shows all established connections for TCP and UDP protocols. `lsof -i` also can be used, but does not have that much possibilities to filter the output. Connections also can be set-up and monitored on an application level (apache, `ftptop`, `ftpwho` for `proftpd`, etc.).

6.1.4 Network protocol

Some protocols are not really optimized for high-bandwidth usage, sometimes a special configuration is needed for sending large amounts of data (e.g. `sendfile` option for `proftpd`). Most of the optimization are discussed in chapter 3.

As this work focuses on servers and server performance, network filesystems are common (also FTP or SFTP, etc., but these are relatively easy to optimize for speed). As for NFS, monitoring and optimization is not that easy, the most used and discussed parameter is the size of a data block that is sent through the network, which should be around 32kiB these days. For more information about NFS optimization, see [26].

6.1.5 Swapping

Although this scenario manifests itself by a high disk usage, the source of the problem is in the memory layer. When thrashing occurs (being the worst case scenario), every process is affected, the kernel utilizes most of the time by moving pages from disk to memory and back. To determine whether virtual memory is full, check the output of `free` on the `Swap:` line.

Only if the memory is mostly used, thrashing can occur, otherwise the swapping is not that intense and is determined by the amount of free physical memory. To really know how much swapping is going on, swap partition data transfers have to be monitored. Memory

monitoring tool `vmstat` gives information about swapping in and out by reading and writing data from and to the swap partition. Of course disk monitoring for the partition itself gives the transfer rates as well.

Using `top` and sorting the output by `VIRT` column (virtual image size – amount of virtual memory used by the process), the highest consumer(s) can be found and then removed from running processes, thus creating more free space in the system.

6.2 High disk usage

When the disk becomes fully utilized, blocking read or write requests take a long time to be serviced, thus slowing the issuing process down. Processes not tied to disk operation continue to function normally. When other possible areas have been tested, the only information that can be retrieved by ordinary tools (tools used in almost every distribution, used on a regular basis) is the affected disk and partition(s).

The overall statistics for partitions is given by `dstat -d -D sda,sdb` (gives output periodically) or `iostat`. This information can solely be enough for some situations, like partitions with restricted access to only few processes. Mostly however it is only the first step to find the affected process and/or file.

Having found the highly used partition, the systemtap script `iotop.stp` should be used. Because SystemTap gathers data from calls to `blktrace` functions, `blktrace` process has to be running. Saving its output is a good practice, it can later be used for more precise analysis. More about storing `blktrace` output in 4.7.2, in this example, `/tmp/boutput` is used. The following commands are needed to get the output from SystemTap (assuming the current directory being the one with script package mentioned in the previous chapter):

```
# btrace /dev/sda >/tmp/boutput &
# stap -I tapsets -DMAXMAPENTRIES=10000 -DMAXACTION=10000 iotop.stp
...
...
Ctrl+C
# fg
Ctrl+C
```

The periodic output gives the overview of the top disk transfer rate consumers. Monitoring of affected files can be done through `topfile.stp`, using the same commands above (changing only the script name) and changing the global variable value inside the script.

6.2.1 Dealing with disk-intensive processes

Now that the affected files and processes are known, actions have to be taken, to decrease the amount of disk usage (either at a local or global scope). This phase gets easier if the administrator knows exactly what processes are running on the server and what exactly should they be doing.

The fastest, but worst, solution is to stop the process from accessing the disk. Killing the process is not really an option (especially on a server), but stopping it for a short while to let other processes finish their work can sometimes be appropriate. Having a backup process or kernel compilation running while a vital data have to be sent to multiple clients is just the case when `SIGSTOP` is an option.

Similar technique is the use of `ionice` tool, although it is available only when the CFQ scheduler is selected. Setting a low priority for non-vital processes resolves the slowdown, but just like stopping said process, it is still just a local solution. In special cases however, it can be used in long-term.

If the disk-intensive process cannot be restricted in any way, further knowledge is needed. The process can read in an unoptimized way, that means reading a large amount of data by relatively small chunks (bytes or tenths of bytes), thus using a system call too many times and forcing the kernel to seek and copy data over and over again. Aside from studying the source code, this information can be obtained from `iostat` (comparing the read/write speed against transactions per second) or `blktrace` output that shows the data size requested (systemtap scripts `spectest.stp` or `traceread.stp` can also be used).

Previously mentioned scripts can also show that the process reads relatively big blocks, but the data is positioned on distant addresses. This can be due to big file fragmentation or more open files in the process which are not adjacent. Moving the files closer together can resolve the problem (see next section for more information about fragmented directories).

The process however can read in an optimized way, but just needs so much data, that the system cannot cope with it. Optimizing the system and setting the program configuration can make a difference (see chapter 3). Hardware upgrade or disk architecture change is appropriate when all previous steps failed.

6.3 Badly positioned files

In this situation, the disk bandwidth is not really utilized, but still, the data retrieval is slow. There are more possible reasons for this, filesystem can be too fragmented or there are many processes accessing different places on the disk, in both cases causing high seek times. Other reasons can be streaming writes issued by one process and chunky reading by another (like copying a large file and browsing a directory).

Basic information is given by `topfile.stp` output. Resolving the most frequently read/written files and their locations can be useful. More exact information is however obtained by the `spectest.stp` script, which reacts on long data retrieval time (long seek or full queue) and prints out affected data locations. Based on the sector numbers, files stored far from each other and usually being read in parallel (database files, web pages), should be moved to be closer (simple file copy should suffice, given that there is enough free space on the device to prevent another fragmentation).

Even more detailed information about the data retrieval process is given by the systemtap script `traceread.stp`. In the script output the administrator can find data about the exact layer of the I/O subsystem where the slowdown occurs (in the queue, when the disk retrieves the data, in copying the data to user, etc).

Fragmentation of a volume is something to be monitored at a regular basis. The tools for getting the fragmentation value and defragmenting are in most cases part of a filesystem-tools package:

- XFS – `xfs_db` – filesystem debugger, the `frag` command retrieves fragmentation level
- ext2/ext3 – `defrag` – a python script for defragmenting
- JFS – `defragfs`
- ReiserFS – doesn't have such tool, although fragmentation is a problem

The basic and most successful way to defragment a volume is to simply copy the data into another volume (and back if needed). For some filesystems it is also the only way to keep filesystem fragmentation in shape.

Aside from using SystemTap scripts to find badly positioned files, more regular and specialized tool is needed. To ease the recognition of badly positioned files and also directories, I created a command-line tool, **fragmon**, with graphical output to achieve this goal, see next chapter.

Chapter 7

Fragmentation monitoring tool

In this chapter I present a specialized tool, **fragmon**, for fragmentation monitoring. It should ease the system maintenance done at a regular basis.

This utility takes a list of files and/or directories and shows the fragmentation map of associated data blocks. The textual output (aside from warnings or progress messages) is a list of file blocks for every file, listed as intervals of continuous data. **fragmon** also creates an **.png** image file showing used data blocks in the scope of given list of files/directories.

7.1 Preliminaries

7.1.1 Fragmentation

Fragmentation is a state of data when there are multiple chunks of it stored at distant places of a storage media. Current storage media define *internal fragmentation* and *external fragmentation*. The internal fragmentation occurs when the smallest physical data units are not filled entirely, thus creating gaps and wasting space. This type of fragmentation is not the issue of this tool for it is only of small consequence these days.

The external fragmentation stands for a much bigger issue. Externally fragmented data can be distributed on a disk in such a way, that some blocks are on the very beginning of the disk and other on the very end. This forces the disk to seek for a very long distance over and over again, creating huge slowdowns. To prevent this from happening, *defragmentation* should be performed.

7.1.2 Defragmentation

Defragmentation is a process of collecting file data and moving it to a consecutive portion of disk space. There are few tools for this job (most of filesystem-tools packages are shipped with some), but the most used and simple method is just to copy (or move) the data from one directory (or disk) to another one. There has to be enough consecutive space on the disk to prevent another fragmentation, of course.

7.2 Motivation

The problem in running environment is not the defragmentation process, but detecting fragmented data. Some data become fragmented very easily and still do not pose a problem

(like temporary files), other will cause big slowdowns to the system (database or file data on a server). Therefore a tool is needed to analyze files and report the fragmentation.

There is a command-line tool **filefrag** (a part of ext2 filesystem tools) that analyzes a file and reports block addresses of all discontinuous data. The function of this tool is exactly what should be used for said analysis, but the scope of it is too narrow. On a running system, many files are read, not only one, the appropriate tool should analyze a given list of files or directories and show an aggregated fragmentation.

Another functionality that is missing, is a more comprehensive output. Textual data of file discontinuities are difficult to follow, graphical output would be of better use to the administrator, giving an overview of data blocks right away.

These two drawbacks of the **filefrag** utility are main motivations for creating a new fragmentation-monitoring utility – **fragmon**.

7.3 Implementation

As mentioned earlier, **filefrag** is a great utility for analyzing one file. According to its license (GPLv2), its source code has been used as a cornerstone of my utility. The system part using `ioctl()` for getting file block map and block addresses stayed and more code implementing directory traversal and data gathering has been added (the original tool only displayed recovered addresses and didn't store any for later use).

The main datasource for statistics and graphical output is the so-called *block map* of a file. Each file is stored as a list of data blocks (the block-size and implementation of the list itself is dependant on a given filesystem). Each data block can be accessed from userspace by its block number, therefore an `ioctl(FIGETBSZ)` call is needed to get the number of blocks in a file.

To really know where the data block is stored on the physical disk, a *linear block-address* (LBA) is needed. Block number is converted to LBA using an `ioctl(FIBMAP)` call. Going through all data blocks of a file and detecting consecutive areas of data is how the program fills the statistical table.

The utility has then been enhanced with a graphical output using **libpng** as a graphical library to produce comprehensive images. Sample output can be found in section 7.4.

As for the directory traversal, only regular files and directories are accepted, the program does not follow symlinks and does not enter directories located on another partitions (the partition is set by the first file/directory of the list to scan).

7.3.1 Output format

As for the statistical output, it is just a listing of consecutive data intervals. The graphical information format is a map of data blocks displayed as cells aligned in a table (much like graphical defragmenting utilities known from Windows OSes). The table has 255 rows and 255 columns, each cell can contain one or more blocks, depending on the overall size of measured data. This ensures that the table stays the same for any data, only the scale changes.

When the overall data span (the interval between the lowest and the highest block address of all given data) is too high to fit in the table, aggregation on blocks is done, so one cell is not only one block, but a group of n blocks. The aggregation value is reported in terminal output.

On the left of the table, the scale is displayed for better head-to-head comparison (the data span can immediately be seen).

The original goal of the tool's output format was to display the exact position of data blocks on physical disk (sectors on a circular disk platter). Such output would be the best for determining good or bad data position, because some data can be relatively far from one another in LBA addressing, but still quickly accessible because of the disk geometry (which is reflected in the C/H/S addressing).

This would require C/H/S addresses instead of LBA ones. Requiring C/H/S address is currently not possible on modern computers, because disk manufacturers (and also the disk hardware) report values that are conforming to addressable space but not to the real low-level formatting of the disk itself. The specification on a disk label (sticker) is also misleading.

7.4 Sample output

Following two pictures both show a block map for the same 1.2GiB ISO image. In the first picture (figure 7.1), the image is on its original disk and it can be seen, that the data is widely spread over the disk. Also the data span reported by the utility is larger (9765378...10251502). The second block map (figure 7.2) was taken after moving the file to another disk (with 50 % free space).

Resulting block span is smaller (1064690...1414794) and most importantly the change can immediately be seen in the picture. Each picture has a different aggregation (8 and 6 respectively), which is reflected on the scale.

For badly fragmented directories (not the file data but whole files are too far from each other), the process is more visible, see 7.3 and 7.4 for comparison. Resulting change is really huge, the block aggregation for the first image is 159 blocks per cell and for the second one, it is only 14 blocks per cell and output images speak for themselves.

One final word about the output images. You can see some *artifacts* in the pictures that seem to be something like a bad alignment of blocks. The reason for this is the filesystem property (not used in filesystems on Windows) that tries to eliminate fragmentation. When a file is created, it is not placed right after another file, but rather few blocks away. This prevents fragmenting due to appending new data to a file. On FAT for example, such added data block is placed on a first free space found and the file is split. In GNU/Linux the data can be appended easily without creating fragments. A really great visual explanation can be found at [24].

7.5 Usage

fragmon takes parameters in a standard UNIX way (it has been implemented using getopt). Switches are: **-v** or **-vv** for verbose (or very verbose) mode, **-s** for printing statistics, **-r** for enabling directory traversal recursively (not only one level as default) and **-o** followed by a filename to specify the output filename for the **.png** image (default value is **out.png**). The rest of command line arguments is taken as a list of input files and directories to analyze.

Like the preceding **filefrag** utility, **fragmon** needs root privileges to access file block maps.

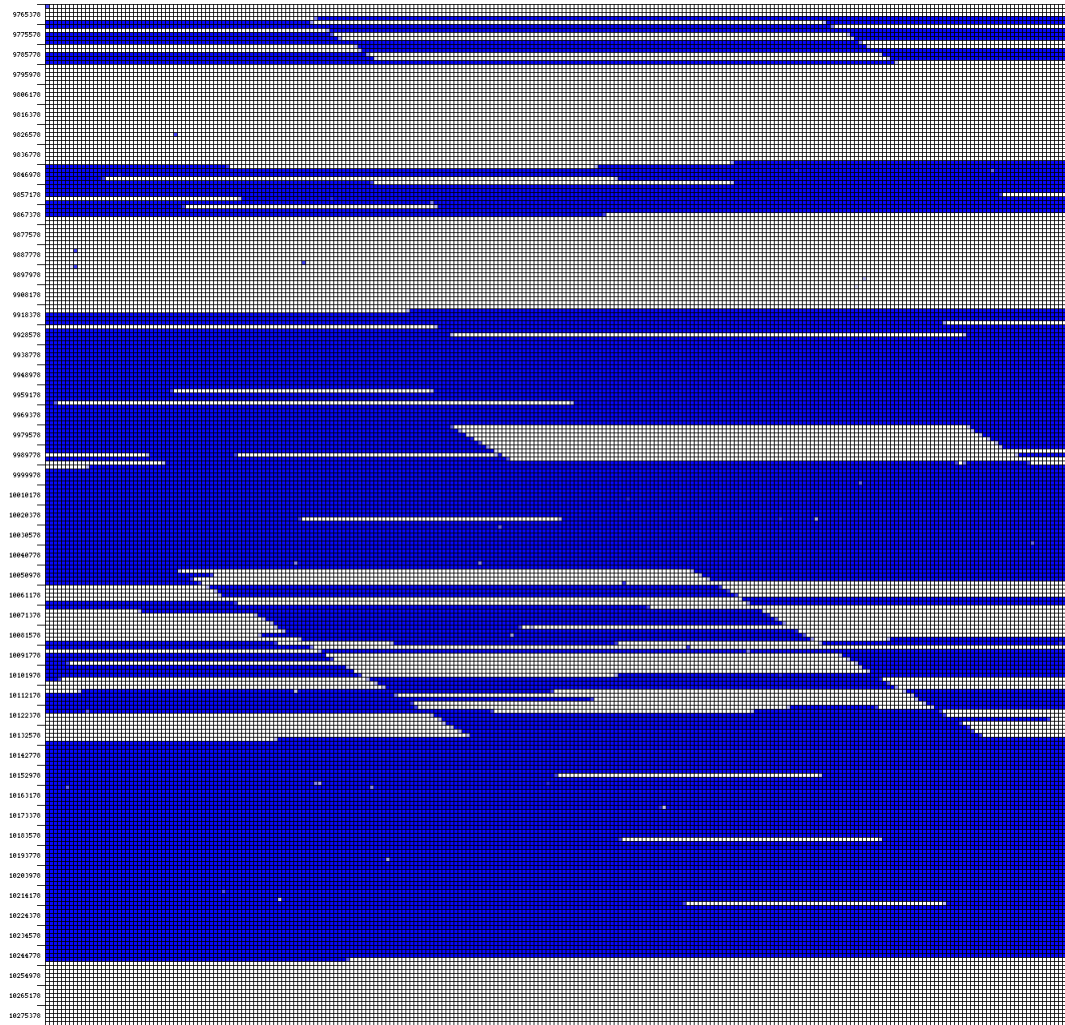


Figure 7.1: File block map before copying

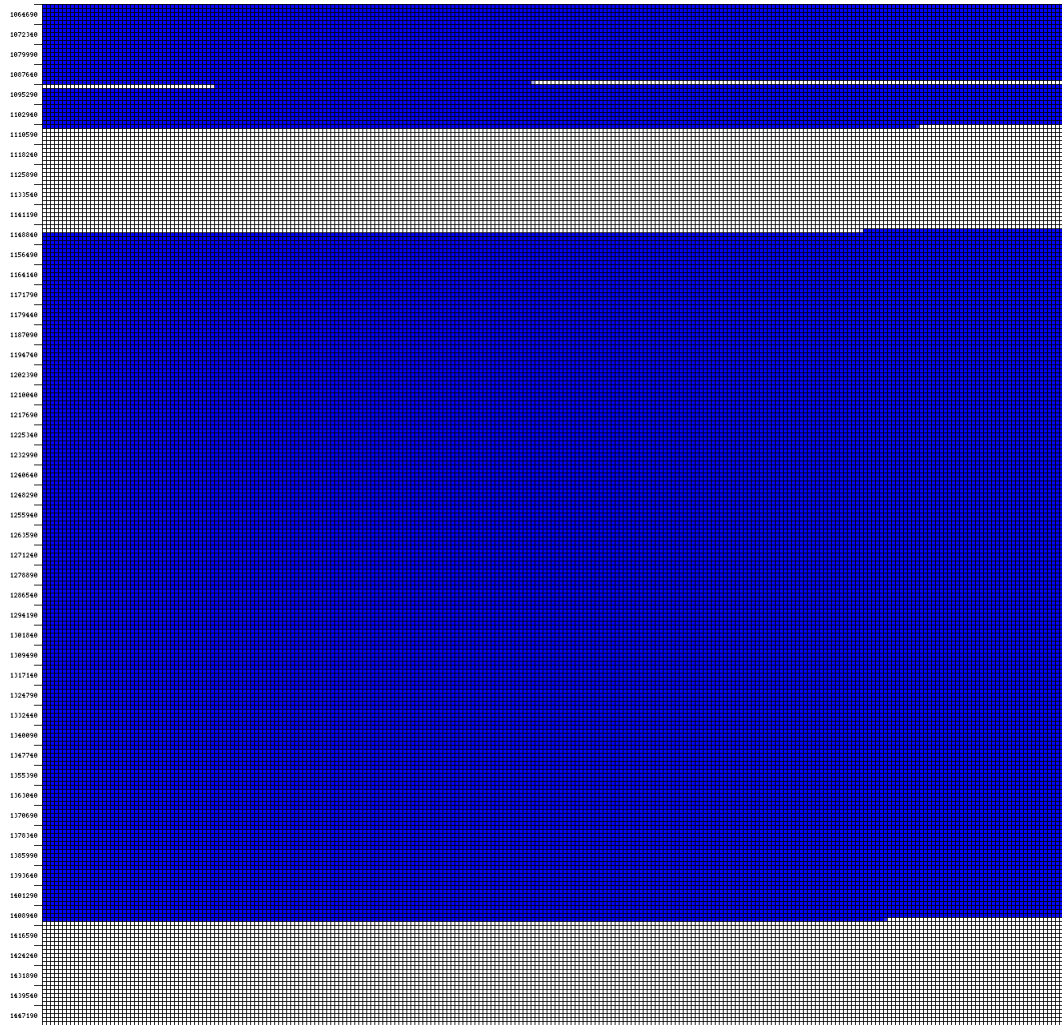
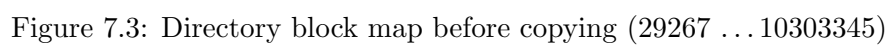


Figure 7.2: File block map after copying



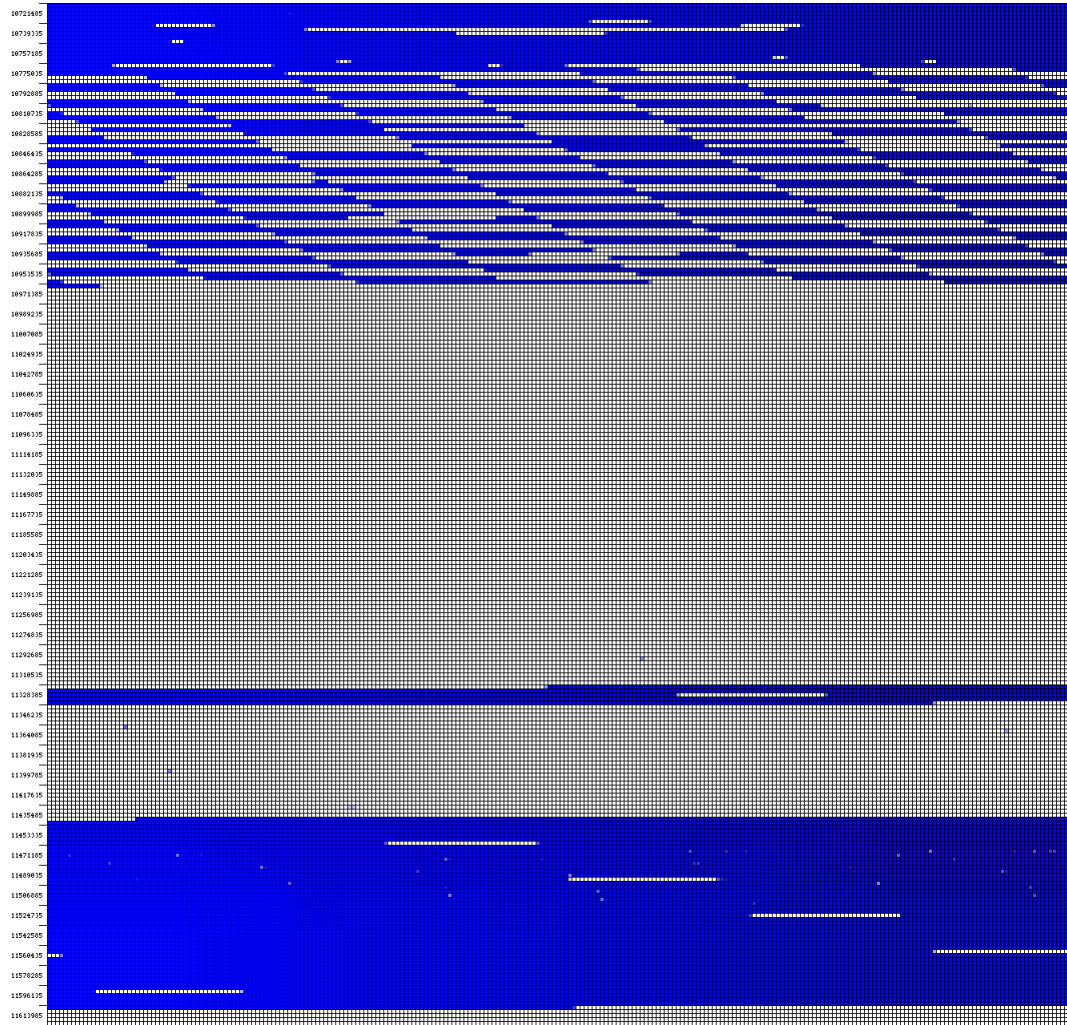


Figure 7.4: Directory block map after copying (10721485 ... 11615829)

Chapter 8

Conclusion

This work has presented the I/O architecture of a Linux kernel. After studying the structure and layers of I/O operations, ways for optimization have been shown and some example settings given. This area is also covered by many other studies, articles and reports, so there should be no problem for the administrators to set the system infrastructure for optimal performance.

Performance optimization can be done in many ways, because there are many layers of the datapath itself and optimizing those layers to cooperate in the best way gives a big performance boost. However initial optimization is only the first step for a well-performing system, which can be done without the system being under a workload. The second and more demanding step follows – monitoring, analyzing and dynamically changing the running system, which in case of servers can be under continuous and heavy workload.

Monitoring the I/O performance on is still somehow problematic. Many tools (as shown) give only general view of the current state of the system, to get more specific data, block I/O tracing has to be used (as the only way to get exact information per file/process basis). The **blktrace** and related programs report all necessary information about I/O requests and queues, but the output form is only usable for backtracing or overall performance of a process.

The solution for such problem is a tracing subsystem – SystemTap. The SystemTap tracing subsystem uses current kernel’s debug information and Kprobes (kernel instrumentation tool) to connect to a running kernel and react on events happening on the fly. By connecting to various function calls (system reads, writes, . . . , but mostly the `__blk_add_trace()`) it gathers information from the running kernel and then formats and parses it to present compact and comprehensive output information.

To ease the monitoring process some more, a specialized fragmentation monitoring tool has been implemented. Based on a **filefrag** tool (using `ioctl()` calls), file and directories are read to create a data block map visualization for a quick and easy monitoring. Administrators then can maintain disk access time through defragmentation if needed.

The thesis presented useful methods and tools to be applied on many situations of system slowdown, particularly focusing on the disk-oriented problems. Using the **blktrace** and SystemTap tools, a new methodology is available to administrators of GNU/Linux systems and hereby implemented **fragmon** eases the monitoring some more. Monitoring the I/O subarchitecture is now becoming less difficult, much like CPU, RAM or network is now.

8.1 Future work

Kernel tracing in general is a very current topic these days. Many changes to the Linux kernel done recently reflex the need of a sophisticated tracing tool for administrators (the addition of probes, block layer tracing, etc.).

The community also tries to create a framework that uses the kernel-side tracing capabilities to ease maintaining and using tracing modules (like previously mentioned SystemTap). SystemTap (and its script libraries located on the project homepage) is quickly developing and it can be predicted that more scripts will be focused on the I/O subarchitecture.

Another Linux kernel tracer is being created at the time. LTTng and output viewer LTTv are being programmed and Linux kernel patches are available. It uses a slightly different approach, using statistical kernel markers. According to the homepage¹, this system should be more comprehensive than SystemTap because of special viewer and XML support. The actual pros and cons of this toolkit will however be known in the future, but due to a different approach it could be a great addition to monitoring possibilities.

Another project in development is a new tracing tool from Intel called LatencyTOP. This year the first version 0.1 has been released and kernel patches are still unstable for normal use. From what can be found on the home website², this project is slightly more focused on desktop systems and at the time it only allows you to use a command-line statistics viewer.

Taking in account, that this work is focused on Linux-based operating systems, most of given methods and tools can also be used on *BSD systems. Unfortunately SystemTap and blktrace are not a part of BSD kernels and are therefore unusable. On the other hand, BSD systems use a similar tracing subsystem called *DTrace*. It is a dynamic tracing system developed by Sun Microsystems, but it has been ported for BSD with some limitations. More information can be found at [22] and [23].

¹<http://ltt.polymtl.ca/>

²<http://www.latencytop.org>

Bibliography

- [1] udev.
<http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html> (20.12.2007)
- [2] Kernel Asynchronous I/O (AIO) Support for Linux.
<http://lse.sourceforge.net/io/aio.html> (20.12.2007)
- [3] Understanding the Linux Kernel, 3rd Edition. *Daniel P. Bovet, Marco Cesati*.
<http://www.linux-security.cn/ebooks/ulk3-html/0596005652/toc.html> (20.12.2007)
- [4] Kernel Korner - I/O Schedulers. *Robert Love*.
<http://www.linuxjournal.com/article/6931> (20.12.2007)
- [5] Choosing an I/O Scheduler for Red Hat® Enterprise Linux® 4 and the 2.6 Kernel.
D. John Shakshober.
<http://www.redhat.com/magazine/008jun05/features/schedulers/> (20.12.2007)
- [6] A Comparison of I/O Schedulers. *Mulyadi Santosa and Fawad Lateef*.
<http://www.samag.com/documents/s=10131/sam0707b/0707b.htm> (20.12.2007)
- [7] Linux Kernel Tuning Using System Control. *Dustin Puryear*
<http://www.samag.com/documents/s=8920/sam0311a/0311a.htm> (20.12.2007)
- [8] Kernel optimization – /etc/sysctl.conf. *Frank Marsh*
<http://frankmash.blogspot.com/2005/11/sysctl-kernel-optimization.html> (11.03.2008)
- [9] Comparison of file systems – Wikipedia, the free encyclopedia.
http://en.wikipedia.org/wiki/Comparison_of_file_systems (20.12.2007)
- [10] Benchmarking Filesystems Part II. *Justin Piszcz*.
<http://linuxgazette.net/122/TWDT.html#piszcz> (20.12.2007)
- [11] Filesystems (ext3, reiser, xfs, jfs) comparison on Debian Etch.
<http://www.debian-administration.org/articles/388> (20.12.2007)
- [12] Block I/O Layer Tracing. *Alan D. Brunelle*.
http://www.gelato.org/pdf/apr2006/gelato_ICE06apr_blktrace_brunelle_hp.pdf (20.12.2007)

- [13] blktrace and blkparse user guide
<https://secure.engr.oregonstate.edu/wiki/CS411/images/2/25/Blktrace.pdf> (04.02.2007)
- [14] Debugfs
<http://lwn.net/Articles/115405/> (20.12.2007)
- [15] SystemTap
<http://sourceware.org/systemtap/> (04.02.2008)
- [16] Deployment Guide – Red Hat Enterprise Linux
http://www.redhat.com/docs/manuals/enterprise/RHEL-5-manual/Deployment_Guide-en-US/ch-systemtap.html (04.02.2008)
- [17] redhat.com – SystemTap
<http://www.redhat.com/magazine/011sep05/features/systemtap/> (04.02.2008)
- [18] relayfs home page
<http://relayfs.sourceforge.net/> (04.02.2008)
- [19] A blktrace tapset, or 101 things you can do with blktrace and systemtap. *Tom Zanussi*
<http://sourceware.org/ml/systemtap/2007-q1/msg00485.html> (04.02.2008)
- [20] /proc/stat explained
<http://www.linuxhowtos.org/System/procstat.htm> (05.02.2008)
- [21] Linux kernel documentation
<kernel source>/Documentation,
<http://www.kernel.org/doc/>
- [22] DTrace – Wikipedia, the free encyklopedia
<http://en.wikipedia.org/wiki/DTrace> (21.02.2008)
- [23] DTrace for FreeBSD
<http://dtrace.what-creek.com/> (21.02.2008)
- [24] OneAndOneIs2 - Why doesn't Linux need defragmenting?
http://geekblog.oneandoneis2.org/index.php/2006/08/17/why_doesn_t_linux_need_defragmenting (21.02.2008)
- [25] Software optimization resources
<http://frankmash.blogspot.com/2005/11/sysctl-kernel-optimization.html> (11.03.2008)
- [26] Optimizing NFS performance
<http://tldp.org/HOWTO/NFS-HOWTO/performance.html> (13.03.2008)

Appendix A

I/O tools outputs

A.1 ifstat

```
a04-0232a ~ # ifstat -tT -i eth0,eth1
```

Time	eth0		eth1		Total	
HH:MM:SS	KB/s in	KB/s out	KB/s in	KB/s out	KB/s in	KB/s out
17:32:01	309.54	12025.43	291.83	11551.86	601.36	23577.29
17:32:02	302.09	11939.68	315.88	11505.03	617.97	23444.71
17:32:03	303.84	12000.47	327.39	11485.13	631.24	23485.60
17:32:04	305.78	12003.00	327.23	11483.94	633.01	23486.94
17:32:05	316.18	12025.31	227.73	6447.10	543.91	18472.41
17:32:06	305.41	11798.82	0.55	1.17	305.96	11799.98
17:32:07	308.70	11968.05	0.00	0.00	308.70	11968.05
17:32:08	128.57	5309.63	0.00	0.00	128.57	5309.63
17:32:09	38.75	1947.94	0.00	0.00	38.75	1947.94
17:32:10	40.71	2022.03	0.00	0.00	40.71	2022.03
17:32:11	40.81	2027.85	0.00	0.00	40.81	2027.85

A.2 dstat

```
a04-0232a ~ # dstat -N eth0,eth1,total -D sda,sdb
```

```
--dsk/sda-----dsk/sdb-- --net/eth0----net/eth1---net/total- ---system--
 read writ: read writ| recv send: recv send: recv send| int  csw
1885k 10k:1885k 9.8k| 0 0 : 0 0 : 0 0 |7237 1554
5772k 0 :5657k 0 | 308k 12M: 0 0 : 308k 12M| 11k 1506
5748k 8192B:5864k 0 | 310k 12M: 0 0 : 310k 12M| 11k 1559
5645k 0 :5748k 4096B| 301k 11M: 0 0 : 301k 11M| 11k 1441
5875k 0 :5645k 0 | 308k 12M: 0 0 : 308k 12M| 11k 1389
5760k 0 :5760k 0 | 311k 12M: 0 0 : 311k 12M| 11k 1486
4109k 0 :4170k 0 | 221k 8802k: 0 0 : 221k 8802k|7972 1272
 897k 20k:1024k 8192B| 44k 2087k: 0 0 : 44k 2087k|1676 678
1024k 0 : 897k 0 | 38k 2026k: 0 0 : 38k 2026k|1481 761
```

A.3 netstat

```
a04-0232a ~ # netstat -A ip -p
Active Internet connections (w/o servers)
Local Address      Foreign Address    State      PID/Program name
a04-0232a:42469    64.12.26.124:aol   ESTABLISHED 6496/sim
a04-0232a:36114    nf-in-f109.google:pop3s TIME_WAIT   -
a04-0232a:41122    java.kn.vutbr.cz:imap ESTABLISHED 6551/kmailfk2Jzb.sl
a04-0232a:ftp      pluto.kn.vutbr.cz:56572 TIME_WAIT   -
a04-0232a:56363    eva.fit.vutbr.cz:imaps ESTABLISHED 6552/kmail5oUrib.sl
a04-0232a:ftp      b07-901b.kn.vutbr.:1267 FIN_WAIT2   -
```

```
a04-0232a ~ # netstat -g
IPv6/IPv4 Group Memberships
Interface      RefCnt Group
-----
eth0            1      ALL-SYSTEMS.MCAST.NET
eth1            1      ALL-SYSTEMS.MCAST.NET
lo              1      ALL-SYSTEMS.MCAST.NET
```

```
a04-0232a ~ # netstat -s
Ip:
  102316833 total packets received
  76759755 forwarded
  0 incoming packets discarded
  25556785 incoming packets delivered
  128652730 requests sent out
```

```
Icmp:
  0 ICMP messages received
  0 input ICMP message failed.
  ICMP input histogram:
  691 ICMP messages sent
  0 ICMP messages failed
  ICMP output histogram:
    destination unreachable: 691
```

The output of `netstat -s` has been truncated and first three columns of `netstat -A ip -p` have been removed.

A.4 lsof

```
a04-0232a ~ # lsof -i
COMMAND  PID  USER  FD  TYPE DEVICE SIZE NODE NAME
dnsmasq  5746 nobody 5u  IPv4  7507      UDP *:bootps
dnsmasq  5746 nobody 6u  IPv4  7512      UDP *:domain
dnsmasq  5746 nobody 7u  IPv4  7513      TCP *:domain (LISTEN)
dnsmasq  5746 nobody 11u IPv4  7523      UDP *:32768
portmap  5854  bin   4u  IPv4  7691      UDP *:sunrpc
```

```

portmap    5854    bin      5u  IPv4    7711      TCP *:sunrpc (LISTEN)
rpc.statd  5918    nobody   6u  IPv4    7835      UDP *:1006
rpc.statd  5918    nobody   8u  IPv4    7843      UDP *:32769
rpc.statd  5918    nobody   9u  IPv4    7846      TCP *:44051 (LISTEN)
rpc.mount  5983     root     6u  IPv4    7945      UDP *:32770
rpc.mount  5983     root     7u  IPv4    7950      TCP *:42309 (LISTEN)
proftpd    6149  proftpd  0u  IPv4    8261      TCP *:ftp (LISTEN)
sshd       6212     root     3u  IPv4    8431      TCP *:ssh (LISTEN)
sim        6496  plague   13u IPv4    9502      TCP
a04-0232a.kn.vutbr.cz:42469->64.12.26.124:aol (ESTABLISHED)
sim        6496  plague   14u IPv4    9483      UDP *:32776
sim        6496  plague   15u IPv4    9497      TCP *:22293 (LISTEN)
proftpd    16658    kn      15u IPv4   34590      TCP
a04-0232a.kn.vutbr.cz:44258->b07-901b.kn.vutbr.cz:1321 (ESTABLISHED)

```

a04-0232a ~ # lsof +d "/tmp/.X11-unix/"

COMMAND	PID	USER	FD	TYPE	DEVICE	SIZE	NODE	NAME
X	6329	root	1u	unix	0xf7b1f880		8651	/tmp/.X11-unix/X0
X	6329	root	20u	unix	0xf687fd00		8848	/tmp/.X11-unix/X0
X	6329	root	21u	unix	0xf668ad80		8934	/tmp/.X11-unix/X0
X	6329	root	22u	unix	0xf6552c80		9369	/tmp/.X11-unix/X0

a04-0232a ~ # lsof -u proftpd

COMMAND	PID	USER	FD	TYPE	DEVICE	SIZE	NODE	NAME
proftpd	6149	proftpd	cwd	DIR	253,11	4096	128	/
proftpd	6149	proftpd	rtd	DIR	253,11	4096	128	/
proftpd	6149	proftpd	txt	REG	253,9	557744	699200	/usr/sbin/proftpd
proftpd	6149	proftpd	0u	IPv4	8261			TCP *:ftp (LISTEN)
proftpd	6149	proftpd	5r	REG	253,11	1454	4196630	/etc/passwd
proftpd	6149	proftpd	6r	REG	253,11	745	4196627	/etc/group

a04-0232a ~ # lsof -p 6496

COMMAND	PID	USER	FD	TYPE	DEVICE	SIZE	NODE	NAME
sim	6496	plague	cwd	DIR	253,12	4096	132	/home/plague
sim	6496	plague	rtd	DIR	253,11	4096	128	/
sim	6496	plague	txt	REG	253,9	14024	162777	/usr/bin/sim
sim	6496	plague	mem	REG	253,9	990428	392694	/usr/lib/sim/icq.so
sim	6496	plague	0r	CHR	1,3		765	/dev/null
sim	6496	plague	5u	unix	0xf58539c0		9450	socket
sim	6496	plague	9w	FIFO	0,5		9457	pipe
sim	6496	plague	14u	IPv4	9483			UDP *:32776
sim	6496	plague	15u	IPv4	9497			TCP *:22293 (LISTEN)

The outputs have been truncated.

A.5 nfsstat

```
xsmrce00@merlin:~$ nfsstat -3
```

Server rpc stats:

calls	badcalls	badauth	badclnt	xdrcll
1179675769	3904	3	3901	0

Server nfs v3:

null	getattr	setattr	lookup	access	readlink
6844406	0%	458488745	38%	4168476	0%
259267773	22%	304602085	25%	585700	0%
read	write	create	mkdir	symlink	mknod
69457303	5%	29353492	2%	5437279	0%
945480	0%	1411	0%	0	0%
remove	rmdir	rename	link	readdir	readdirplus
5359994	0%	1017627	0%	85947	0%
23648	0%	88672	0%	10634022	0%
fsstat	fsinfo	pathconf	commit		
42293	0%	77504	0%	0	0%
21443135	1%				

Client rpc stats:

calls	retrans	authrefrsh
130390105	5132	0

Client nfs v3:

null	getattr	setattr	lookup	access	readlink
0	0%	51364926	39%	3230253	2%
16396169	12%	14629954	11%	361223	0%
read	write	create	mkdir	symlink	mknod
17818431	13%	17167416	13%	1257718	0%
26892	0%	9911	0%	74135	0%
remove	rmdir	rename	link	readdir	readdirplus
2766762	2%	32598	0%	400668	0%
91186	0%	411729	0%	4175304	3%
fsstat	fsinfo	pathconf	commit		
135594	0%	21	0%	0	0%
39194	0%				

xsmrce00@merlin:~\$ nfsstat -m

/homes/eva from eva:/home/users

Flags: rw,nosuid,nodev,v3,rsz=16384,wsz=16384,hard,intr,lock,
proto=tcp,addr=eva

/homes/mail from eva:/var/mail

Flags: rw,nosuid,nodev,v3,rsz=16384,wsz=16384,hard,intr,lock,
proto=tcp,addr=eva

/homes/kazi from kazi:/home/users

Flags: rw,nosuid,nodev,v3,rsz=16384,wsz=16384,hard,intr,lock,
proto=tcp,addr=kazi

A.6 smartctl

a04-0232a ~ # smartctl -i /dev/sda

smartctl version 5.37 [i686-pc-linux-gnu] Copyright (C) 2002-6 Bruce Allen

Home page is <http://smartmontools.sourceforge.net/>

=== START OF INFORMATION SECTION ===

Model Family: Western Digital Caviar SE16 family
Device Model: WDC WD5000KS-00MNB0
Serial Number: WD-WCANU1544624
Firmware Version: 07.02E07
User Capacity: 500 107 862 016 bytes
Device is: In smartctl database [for details use: -P show]
ATA Version is: 7
ATA Standard is: Exact ATA specification draft version not indicated
Local Time is: Thu Dec 20 18:31:35 2007 CET
SMART support is: Available - device has SMART capability.
SMART support is: Enabled

a04-0232a ~ # smartctl -A /dev/sda

smartctl version 5.37 [i686-pc-linux-gnu] Copyright (C) 2002-6 Bruce Allen
Home page is <http://smartmontools.sourceforge.net/>

=== START OF READ SMART DATA SECTION ===

SMART Attributes Data Structure revision number: 16

Vendor Specific SMART Attributes with Thresholds:

ID#	ATTRIBUTE_NAME	FLAG	VALUE	WORST	THRESH	TYPE	UPDATED
1	Raw_Read_Error_Rate	0x000f	200	200	051	Pre-fail	Always
3	Spin_Up_Time	0x0003	224	224	021	Pre-fail	Always
4	Start_Stop_Count	0x0032	100	100	000	Old_age	Always
5	Reallocated_Sector_Ct	0x0033	200	200	140	Pre-fail	Always
7	Seek_Error_Rate	0x000f	200	200	051	Pre-fail	Always
9	Power_On_Hours	0x0032	093	093	000	Old_age	Always
10	Spin_Retry_Count	0x0013	100	100	051	Pre-fail	Always
11	Calibration_Retry_Count	0x0012	100	100	051	Old_age	Always
12	Power_Cycle_Count	0x0032	100	100	000	Old_age	Always
194	Temperature_Celsius	0x0022	112	104	000	Old_age	Always
196	Reallocated_Event_Count	0x0032	200	200	000	Old_age	Always
197	Current_Pending_Sector	0x0012	200	200	000	Old_age	Always
198	Offline_Uncorrectable	0x0010	200	200	000	Old_age	Offline
199	UDMA_CRC_Error_Count	0x003e	200	200	000	Old_age	Always
200	Multi_Zone_Error_Rate	0x0009	200	200	051	Pre-fail	Offline

A.7 iostat

xsmrce00@PC0104-15:~\$ /usr/bin/iostat -n

Linux 2.6.22.14 (PC0104-15.fit.vutbr.cz)

20.12.2007

avg-cpu:	%user	%nice	%system	%iowait	%steal	%idle
	0,01	0,00	0,03	0,05	0,00	99,91

Device:	tps	Blk_read/s	Blk_wrtn/s	Blk_read	Blk_wrtn
sda	1,10	24,52	16,59	757762	512782

```
Device:   rBlk_nor/s wBlk_nor/s rBlk_dir/s wBlk_dir/s rBlk_svr/s wBlk_svr/s
merlin:/root    0,25      0,00      0,00      0,00      0,15      0,00
merlin:/pub     0,00      0,00      0,00      0,00      0,00      0,00
```

```
a04-0232a ~ # iostat
Linux 2.6.23-gentoo-r3-plague (a04-0232a)      20.12.2007
```

```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0,99    0,00    4,34    0,45    0,00   94,22
```

```
Device:            tps    Blk_read/s    Blk_wrtn/s    Blk_read    Blk_wrtn
sda                 96,53      3642,76      18,40    89385875    451560
sdb                 96,70      3642,90      17,31    89389166    424800
```

```
a04-0232a ~ # iostat -x
Linux 2.6.23-gentoo-r3-plague (a04-0232a)      20.12.2007
```

```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0,99    0,00    4,34    0,45    0,00   94,22
```

```
Dev: rrqm/s wrqm/s   r/s   w/s   rsec/s wsec/s rq-sz qu-sz await svctm %util
sda 3533,46   0,43 95,93 0,67 3645,31 18,41 37,93 0,12 1,21 0,68 6,60
sdb 3533,24   0,48 96,17 0,60 3645,45 17,32 37,85 0,12 1,21 0,68 6,56
```

A.8 blktrace

```
a04-0232a ~ # btrace /dev/sda
8,0 1 1 0.000000000 3033 A W 875394956 + 8 <- (253,0) 1750789772
8,0 1 2 0.000000441 3033 Q W 875394956 + 8 [xfsbufd]
8,0 1 3 0.000001588 3033 G W 875394956 + 8 [xfsbufd]
8,0 1 4 0.000002286 3033 P N [xfsbufd]
8,0 1 5 0.000002611 3033 I W 875394956 + 8 [xfsbufd]
8,0 1 6 0.000037499 3033 D W 875394956 + 8 [xfsbufd]
8,0 0 1 0.481468667 0 C W 875394956 + 8 [0]
8,0 0 2 1.383417343 6499 A R 906697172 + 16 <- (253,0) 1813394132
8,0 0 3 1.383417661 6499 Q R 906697172 + 16 [krusader]
8,0 0 4 1.383419021 6499 G R 906697172 + 16 [krusader]
8,0 0 5 1.383419588 6499 P N [krusader]
8,0 0 6 1.383419943 6499 I R 906697172 + 16 [krusader]
8,0 0 7 1.383422640 6499 D R 906697172 + 16 [krusader]
8,0 0 8 1.394178391 0 C R 906697172 + 16 [0]
```

```
CPU0 (8,0):
```

```
Reads Queued:      1,      8KiB Writes Queued:      0,      0KiB
Read Dispatches:   1,      8KiB Write Dispatches:    0,      0KiB
Reads Requeued:    0                Writes Requeued:    0
Reads Completed:   1,      8KiB Writes Completed:    1,      4KiB
Read Merges:       0,      0KiB Write Merges:        0,      0KiB
Read depth:        1                Write depth:      1
```

```

IO unplugs:          0          Timer unplugs:          0
CPU1 (8,0):
Reads Queued:        0,      0KiB  Writes Queued:        1,      4KiB
Read Dispatches:     0,      0KiB  Write Dispatches:   1,      4KiB
Reads Requeued:      0          Writes Requeued:      0
Reads Completed:     0,      0KiB  Writes Completed:   0,      0KiB
Read Merges:         0,      0KiB  Write Merges:       0,      0KiB
Read depth:          1          Write depth:        1
IO unplugs:          0          Timer unplugs:          0

```

```

Total (8,0):
Reads Queued:        1,      8KiB  Writes Queued:        1,      4KiB
Read Dispatches:     1,      8KiB  Write Dispatches:   1,      4KiB
Reads Requeued:      0          Writes Requeued:      0
Reads Completed:     1,      8KiB  Writes Completed:   1,      4KiB
Read Merges:         0,      0KiB  Write Merges:       0,      0KiB
IO unplugs:          0          Timer unplugs:          0

```

Throughput (R/W): 5KiB/s / 2KiB/s

Events (8,0): 14 entries

Skips: 0 forward (0 - 0.0%)

a04-0232a ~ # blktrace -d /dev/sda -o test.blktrace

Device: /dev/sda

```

CPU 0:          0 events,          4 KiB data
CPU 1:          0 events,          2 KiB data
Total:          0 events (dropped 0),      5 KiB data

```

a04-0232a ~ # btt -i test.blktrace.blktrace.1

===== All Devices =====

	ALL	MIN	AVG	MAX	N
Q2Q	0.000002862	0.000008885	0.000024049		6

===== Device Overhead =====

DEV	Q2I	I2D	D2C
-----	-----	-----	-----

===== Device Merge Information =====

DEV	#Q	#D	Ratio	BLKmin	BLKavg	BLKmax	Total
-----	-----	-----	-----	-----	-----	-----	-----

===== Device Seek Information =====

DEV	NSEEKS	MEAN	MEDIAN	MODE
-----	--------	------	--------	------

```

----- | -----
(8,0) |      6 151070921.3      0 | 6(1) 21 9266519 9266488 875394956

```

===== Plug Information =====

```

      DEV |      # Plugs # Timer Us | % Time Q Plugged
----- | -----

```

Total System

Total System : q activity

25009.139355413 0.0

25009.139355413 0.4

25009.139408724 0.4

25009.139408724 0.0

Total System : c activity

Per device

8,0 : q activity

25009.139355413 1.0

25009.139355413 1.4

25009.139408724 1.4

25009.139408724 1.0

8,0 : c activity

Per process

xfsbufd : q activity

25009.139355413 3.0

25009.139355413 3.4

25009.139408724 3.4

25009.139408724 3.0

xfsbufd : c activity

Appendix B

SystemTap examples

B.1 kprobeio.stp

```
/* kprobeio.stp
   This is a simple module to get information about block I/O operations.
   Will Cohen
*/

global count_generic_make_request

probe kernel.function("generic_make_request")
{
    ++count_generic_make_request;
}

probe begin { log("starting probe") }

probe end
{
    log("ending probe")
    log("generic_make_request() called "
        . string(count_generic_make_request)
        . " times");
}
```

B.2 countall.stp

MAJ	MIN	ACTION	RW	COUNT	KiB
3	0	backmerge	R	4600	21748
3	0	backmerge	W	65545	262180
3	0	backmerge	RM	4	16
3	0	complete	R	8537	104580
3	0	complete	W	13209	316868
3	0	frontmerge	W	36	144

3	0	frontmerge	R	12	52
3	0	get request	R	7620	79096
3	0	get request	W	13636	54544
3	0	get request	RM	916	3664
3	0	insert	R	8536	82760
3	0	insert	W	13636	54544
3	0	issue	R	8536	104576
3	0	issue	W	13209	316868
3	0	plug	R	8621	0
3	0	queue	R	12232	100896
3	0	queue	W	79217	316868
3	0	queue	RM	920	3680
3	0	unplug	R	10031	0
3	0	unplug timer	R	20	0

B.3 spectest.stp

q2c time > 1 second detected, dumping and exiting...

q2c (1.009127993) = q2d (0.995869990) + d2c (0.013258003)

Last 15 records of trace (locate complete data run in blktrace output):

3,0	0	376.050241410	9954	D	W 141509292 + 16 [cc1]
3,0	0	376.063755312	9954	C	W 141509292 + 16 [0]
3,0	0	376.063817162	9954	D	W 141509412 + 8 [cc1]
3,0	0	376.067128381	9954	C	W 141509412 + 8 [0]
3,0	0	376.067182275	9954	D	W 141509436 + 8 [cc1]
3,0	0	376.070480022	9954	C	W 141509436 + 8 [0]
3,0	0	376.070531315	9954	D	W 141509484 + 8 [cc1]
3,0	0	376.072908834	9954	C	W 141509484 + 8 [0]
3,0	0	376.072961535	9954	D	W 141509540 + 8 [cc1]
3,0	0	376.073211216	9954	C	W 141509540 + 8 [0]
3,0	0	376.073240432	9954	D	W 141509604 + 8 [cc1]
3,0	0	376.073502322	9954	C	W 141509604 + 8 [0]
3,0	0	376.073531807	9954	D	W 141509700 + 8 [cc1]
3,0	0	376.077646354	9954	C	W 141509700 + 8 [0]
3,0	0	376.077702407	9954	D	W 141509716 + 8 [cc1]
3,0	0	376.090960410	9954	C	W 141509716 + 8 [0]

B.4 iotop.stp

PID	EXECNAME	TOTAL(k)	QR(k)	QW(k)
5649	firefox-bin	6388	6388	0
1900	kjournald	400	0	400
5641	cc1	200	200	0
5708	cc1	44	44	0

5617	make	24	24	0
5662	cc1	24	24	0
5472	as	20	0	20
5476	fixdep	20	0	20
5620	cc1	20	20	0
5346	fixdep	16	0	16
5361	fixdep	16	0	16
5384	fixdep	16	0	16
5400	fixdep	16	0	16
5420	fixdep	16	0	16
5430	fixdep	16	0	16
4543	bash	16	16	0
5350	ld	12	0	12

B.5 topfile.stp

INO	FILENAME	TOTAL(k)	QR(k)	QW(k)
-----	-----	-----	-----	-----
458901	XUL.mfasl	896	896	0
1540368	nsExtensionManager.js	268	268	0
1540370	nsUpdateService.js	228	228	0
1655550	_CACHE_002_	160	88	72
1655549	_CACHE_001_	156	96	60
1753153	bookmarks.properties	148	148	0
475249	compreg.dat	144	144	0
1986358	forms.css	124	124	0
1986359	ua.css	116	116	0
1655551	_CACHE_003_	100	48	52

B.6 traceread.stp

```

tracking file module.c
  3.358570676      0.000000000      T R
tracking file module.c, ino 548922
  11.192565429      0.000000000 vfs_read.entry cat(4932) file module.c,
pos 0 count 4096
  11.192612604      0.000047175      Q R sector 120456092, bytes 16384
  11.192639507      0.000026903      G R sector 120456092, bytes 16384
  11.192654697      0.000015190      P R
  11.192666705      0.000012008      I R sector 120456092, bytes 16384
  11.192685104      0.000018399      U R
  11.192701796      0.000016692      D R sector 120456092, bytes 16384
  11.197560999      0.004859203      C R sector 120456092, bytes
16384, q2d 0.000089192, d2c 0.004859203, q2c 0.004948395
  11.197612869      0.000051870 vfs_read.exit cat(4932) file module.c

```