

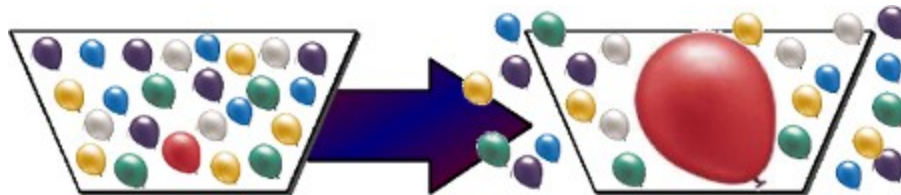
Case study with atop: memory leakage

Gerlof Langeveld & Jan Christiaan van Winkel

www.atoptool.nl

– May 2010 –

This document describes the analysis of a slow system suffering from a process with a memory leakage. Such process regularly requests for more dynamic memory with the subroutine `malloc`, while the programmer has “forgotten” to free the requested memory again. In this way the process grows virtually as well as physically. Mainly by the *physical* growth (the process' resident set size increases), the process inflates like a balloon and pushes other processes out of main memory. Instead of a healthy system where processes reach a proper balance in their memory consumption, the total system performance might degrade just by one process that leaks memory.



Notice that the Linux kernel does not limit the physical memory consumption of processes. Every process, either running under root identity or non-root identity, can grow unlimited. In the last section of this case study, some suggestions will be given to decrease the influence of leaking processes on your overall system performance.

In order to be able to interpret the figures produced by `atop`, basic knowledge of Linux memory management is required. The next sections describe the utilization of physical memory as well as the impact of virtual memory before focussing on the details of the case itself.

Introduction to physical memory

The physical memory (RAM) of your system is subdivided in equally-sized portions, called *memory pages*. The size of a memory page depends on the CPU-architecture and the settings issued by the operating system. Let's assume for this article that the size of a memory page is 4 KiB.

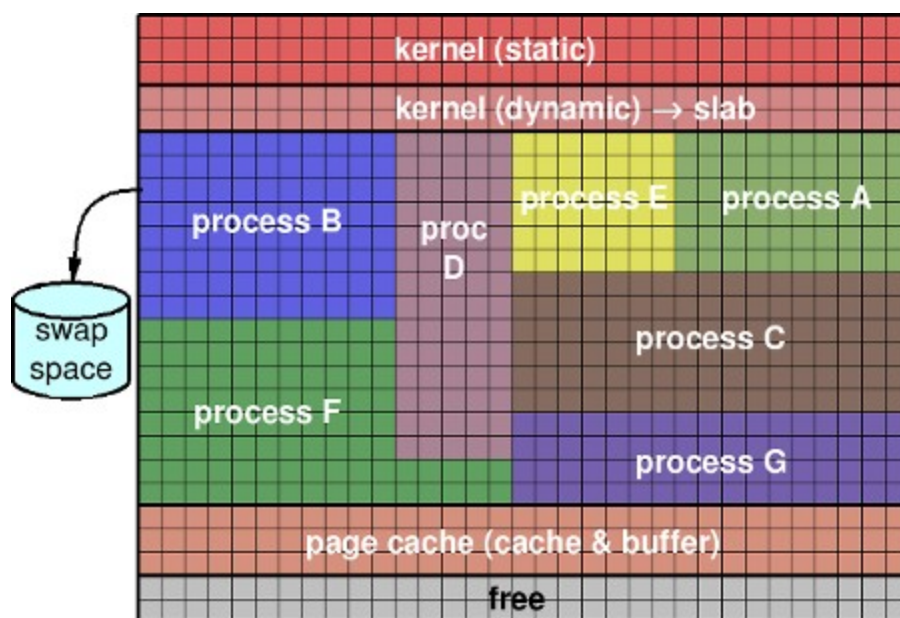
At the moment that the system is booted, the compressed kernel image – known as the file `/boot/vmlinuz- . . .` – is loaded and decompressed in memory. This static part of the kernel is loaded somewhere at the beginning of the RAM memory.

The running kernel requires however more space, e.g. for the administration of processes, open files, network sockets, ... but also to load dynamic loadable modules. Therefore, the kernel dynamically allocates memory using so-called “slab caches”, in short *slab*. When this dynamically allocated kernel memory is not needed any more (removal of process administration when a process exits, unloading a loaded module, ...), the kernel might free this memory. This means that the slab space will shrink again. Notice that all pages in use by the kernel are memory resident and will never be swapped.

Apart from the kernel, also processes require physical pages for their text (code), static data and stack. The physical space consumed by a process is called “Resident Set Size”, in short *RSS*. How a page becomes part of the *RSS* will be discussed in the next section.

The remaining part of the physical memory – after the kernel and processes have taken their share – is mainly used for the *page cache*. The page cache keeps as much data as possible from the disks (filesystems) in memory in order to improve the access speed to disk data. The page cache consists of two parts: the part where the data blocks of files are stored and the part where the metadata blocks (superblocks, inodes, bitmaps, ...) of filesystems are stored. The latter part is called the “buffer cache”. Most tools (like `free`, `top`, `atop`) show two separate values for these parts, resp. “cached” and “buffer”. The sum of these two values is the total size of the page cache.

The size of the page cache varies. If there is plenty free memory, the page cache will grow and if there is a lack of memory, the page cache will shrink again.



Finally, the kernel keeps a pool of free pages to be able to fulfill a request for a new page and deliver it from stock straight away. When the number of free pages drops below a particular threshold, pages that are currently occupied will be freed and added to the free page pool. Such page can be retrieved from a process (current page contents might have to be swapped to swap space first) or it can be stolen from the page cache (current page contents might have to be flushed to the filesystem first). In the first case, the *RSS* of the concerning process shrinks. In the second case, the size of the page cache shrinks.

Even the slab might shrink in case of a lack of free pages. Also the slab contains data that is just meant to speed up certain mechanisms, but can be shrunk in case of memory pressure. An example is the *incore inode cache* that contains inodes of files that are currently open, but also contains inodes of files that have recently been open but are currently closed. That last category is kept in memory, just in case such file will be opened again in the near future (saves another inode retrieval from disk). If needed however, the incore inodes of closed files can be removed. Another example is the *directory name cache* (dentry cache) that holds the names of recently accessed files and directories. The dentry cache is meant to speed up the pathname resolution by avoiding accesses to disk. In case of memory pressure, the least-recently accessed names might be removed to shrink the slab.

In the output of `atop`, the sizes of the memory components that have just been discussed can be found:

ATOP - sammy		2010/03/02 16:14:33		----		10s elapsed	
PRC	sys 0.45s	user 6.12s	#proc 220	#zombie 0	#exit 0		
CPU	sys 4%	user 61%	irq 0%	idle 134%	wait 0%		
CPL	avg1 0.31	avg5 0.34	avg15 0.62	csw 8414	intr 22038		
MEM	tot 3.8G	free 2.3G	cache 322.7M	buff 249.9M	slab 152.8M		
SWP	tot 8.0G	free 7.7G		vmcom 1.8G	vmlim 9.9G		
LVM	vg00-lvhome	busy 1%	read 0	write 5	avio 11.0 ms		
LVM	vg00-lvtmp	busy 0%	read 0	write 31	avio 1.42 ms		
DSK	sda	busy 1%	read 0	write 13	avio 7.62 ms		

In the line labeled MEM, the size of the RAM memory is shown (`tot`), the memory that is currently free (`free`), the size of the page cache (`cache+buff`) and the size of the dynamically allocated kernel memory (`slab`).

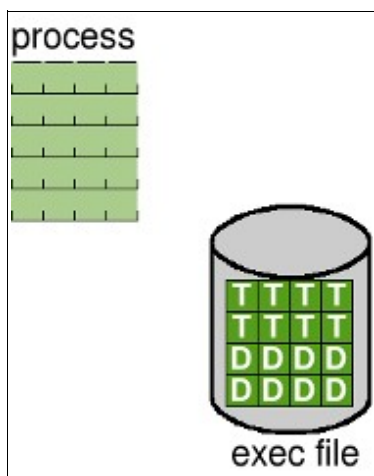
The RSS of the processes can be found in the process list (the lower part of screen):

PID	MINFLT	MAJFLT	VSTEXT	VSIZE	RSIZE	VGROW	RGROW	MEM	CMD	1/2
31186	0	0	2K	259.7M	246.3M	0K	0K	6%	upload	
3456	451	0	3K	1.2G	112.1M	1628K	1596K	3%	simpressexec.bin	
29272	60	0	34499K	1.0G	99396K	0K	240K	2%	chrome	
4680	0	0	74K	749.4M	70140K	-272K	-272K	2%	firefox	
1530	795	0	1778K	174.7M	23180K	6116K	0K	1%	Xorg	

The memory details (subcommand `m`) show the current RSS per process in the column RSIZE and (as a percentage of the total memory installed) in the column MEM. The physical growth of the process during the last interval is shown in the column RGROW.

Introduction to virtual memory

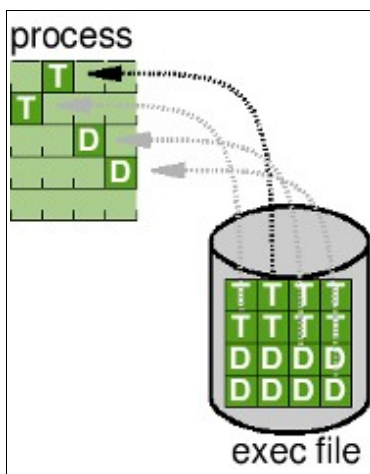
When a new program is activated, the kernel constructs a *virtual* address space for the new process. This virtual address space describes all memory that the process could possibly use. For a starting process, the size of the virtual space is mainly determined by the text (T) and data (D) pages in the executable file, with a few additional pages for the process' stack.



Notice that the process does not consume any physical memory yet during its early startup stage.

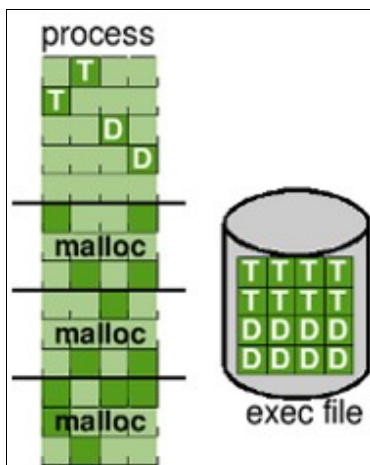
The illustration shows an executable file with 32KiB (8 pages) of text and 32KiB (8 pages) of static data. The kernel has just built a virtual address space for the pages of the executable file and 4 additional pages (e.g. for stack).

After the virtual address space has been built, the kernel fills the program counter register of the CPU with the address of the first instruction to be executed. The CPU tries to fetch this instruction, but



notices that the concerning text page is not in memory. Therefore the CPU generates a fault (trap). The fault handling routine of the kernel will load the requested text page from the executable file and restarts the process at the same point. Now the CPU is able to fetch the first instruction and executes it. When a branch is made to an instruction that lies in another text page, that page will be loaded via fault handling as well. The kernel will load a data page as soon as a reference is made to a static variable. In this way, any page can be physically loaded into memory at its first reference. Pages that are not referenced at all, will not be loaded into memory.

The illustration shows that the process has a *virtual* size of 80KiB (20 pages) and a physical size (RSS) of 16KiB (4 pages). Obviously, the physical size is always a subset of the virtual size and can never be larger than that.



Suppose that the process allocates memory dynamically (with the subroutine `malloc`), the requested space will initially only extend the process' virtual address space. Only when the process really refers to a page in the dynamic area, that page is physically created and filled with binary zeroes.

The illustration shows that the first malloc'ed area has a virtual size of 48KiB (12 pages) while only 16KiB (4 pages) are physically created by a reference. Also the pages in the second and third malloc'ed space have not all been referenced.

In the process list shown by `atop`, information can be found about the virtual address space:

PID	MINFLT	MAJFLT	VSTEXT	VSIZE	RSIZE	VGROW	RGROW	MEM	CMD	1/2
31186	0	0	2K	259.7M	246.3M	0K	0K	6%	upload	
3456	451	0	3K	1.2G	112.1M	1628K	1596K	3%	simpressexec.bin	
29272	60	0	34499K	1.0G	99396K	0K	240K	2%	chrome	
4680	0	0	74K	749.4M	70140K	-272K	-272K	2%	firefox	
1530	795	0	1778K	174.7M	23180K	6116K	0K	1%	Xorg	

The memory details (subcommand `m`) show the current virtual size per process in the column `VSIZE`. The virtual growth of a process during the last interval is shown in the column `VGROW`.

For process `simpressexec.bin` with pid 3456 a virtual growth of 1628KiB is shown (probably by issuing a `malloc`) while the resident growth is 1596KiB. Process `chrome` with pid 29272 has not been grown virtually (0KiB) but has referenced pages (60 pages of 4KiB) during the last interval that have been allocated virtually during earlier intervals. Process `Xorg` with pid 1530 has only been grown virtually (6116KiB) but has not referenced any of the new pages (yet).

Process `firefox` with pid 4680 has *freed* malloc'ed space with a virtual size of 272KiB. Apparently, this implies the release of 272KiB resident space.

Case study: A quiet system

The first `atop` snapshot was taken when a program that leaks memory has just been started, called `lekker` (Dutch for “leaker”). In this snapshot we see a system with net 3.8GiB of physical memory of which 186MiB is free and more than 1GiB in the page cache (MEM line: `cache + buff`). The kernel has dynamically allocated 273MiB slab. Thus, some 2.3GiB is in use by application processes. Swap space (8GiB) is almost unused (so far...). We can see that the `lekker` process has grown 156Mib (virtual) during the last interval which was also made resident by really referencing the allocated space. For now, there is no reason to be worried.

ATOP - sammy		2010/03/02		15:12:12		----		20s elapsed		
PRC	sys	2.46s	user	7.17s	#proc	229	#zombie	0	#exit	0
CPU	sys	12%	user	35%	irq	0%	idle	149%	wait	4%
CPL	avg1	0.45	avg5	0.45	avg15	0.69	csw	52480	intr	40316
MEM	tot	3.8G	free	186.7M	cache	706.8M	buff	324.2M	slab	273.2M
SWP	tot	8.0G	free	7.8G			vmcom	3.4G	vmlim	9.9G
LVM	vg00-lvhome		busy	2%	read	0	write	10	avio	42.5 ms
LVM	vg00-lvtmp		busy	0%	read	0	write	5	avio	12.6 ms
LVM	vg00-lvvar		busy	0%	read	0	write	6	avio	0.00 ms
DSK	sda		busy	2%	read	0	write	21	avio	23.2 ms
NET	transport		tcp	36	tcpo	45	udpi	0	udpo	0
NET	network		ipi	41	ipo	46	ipfrw	0	deliv	36
NET	wlan0	----	pcki	42	pcko	47	si	1 Kbps	so	4 Kbps

PID	MINFLT	MAJFLT	VSTEXT	VSIZE	RSIZE	VGROW	RGROW	MEM	CMD	1/17
30457	0	0	2K	259.7M	256.3M	0K	0K	7%	upload	
30926	0	0	2K	259.7M	256.3M	0K	0K	7%	upload	
30925	0	0	2K	259.7M	256.3M	0K	0K	7%	upload	
30456	0	0	2K	259.7M	256.3M	0K	0K	7%	upload	
30425	0	0	2K	259.7M	256.0M	0K	0K	7%	upload	
30455	0	0	2K	259.7M	252.3M	0K	0K	6%	upload	
31048	40020	0	2K	175.7M	172.3M	156.3M	156.3M	4%	lekker	
29272	143	0	34499K	1.0G	103.8M	0K	-108K	3%	chrome	
3456	0	0	3K	1.1G	91232K	0K	0K	2%	simpres.s.bin	
4680	0	0	74K	640.4M	69532K	0K	0K	2%	firefox	
30574	0	0	34499K	958.3M	52416K	0K	0K	1%	chrome	
30549	46	0	34499K	622.4M	44972K	0K	0K	1%	chrome	
3343	0	0	127K	51148K	38872K	0K	0K	1%	atop	
29211	0	0	34499K	1.5G	30600K	0K	0K	1%	chrome	

Please take notice of the six `upload` processes. They have allocated 256MiB each (virtual and resident). We can also see multiple `chrome` processes that may have a large virtual size, but only a small portion of that is resident: the cumulated virtual size is 4.1GiB (1+1+0.6+1.5 GiB) of which 229MiB (103+52+44+30 MiB) is resident. The process `simpres.s.bin` has made less than 10% of its virtual footprint (1.1GiB) resident (91MiB). Also `firefox` has a relatively small portion of its virtual footprint resident. A lot of these virtual sizes will be shared, not only for the same executable file (4 `chrome` processes share at least the same code), but also for shared library code used by all processes. Till now the system has “promised” 3.4GiB of virtual memory (MEM line, `vmcom`) of the total limit 9.9GiB (`vmlim`, which is the size of swap space plus half of the physical memory).

Case study: It's getting a bit busy...

One snapshot of twenty seconds later, `lekker` has grown another 150MiB (virtual and resident). For a large part that could be claimed from the free space, but not entirely. The number of free pages in stock is getting very low, so the kernel tries to free memory somewhere else. We can see that the first victims are the page cache and the slab. They both have to shrink.

ATOP - sammy		2010/03/02		15:12:32		----		20s elapsed		
PRC	sys	2.72s	user	6.99s	#proc	229	#zombie	0	#exit	0
CPU	sys	13%	user	34%	irq	0%	idle	151%	wait	2%
CPL	avg1	0.48	avg5	0.46	avg15	0.69	csw	48261	intr	38102
MEM	tot	3.8G	free	56.5M	cache	697.7M	buff	315.1M	slab	272.0M
SWP	tot	8.0G	free	7.8G			vmcom	3.5G	vmlim	9.9G
PAG	scan	4180	stall	0			swin	0	swout	0
LVM	vg00-lvhome		busy	2%	read	15	write	111	avio	2.60 ms
LVM	vg00-lvtmp		busy	0%	read	0	write	1	avio	1.00 ms
DSK	sda		busy	2%	read	15	write	54	avio	4.74 ms
NET	network		ipi	3	ipo	1	ipfrw	0	deliv	0
NET	wlan0	----	pcki	6	pcko	2	si	0 Kbps	so	0 Kbps

PID	MINFLT	MAJFLT	VSTEXT	VSIZE	RSIZE	VGROW	RGROW	MEM	CMD	1/16
31048	38019	0	2K	324.2M	320.8M	148.5M	148.5M	8%	lekker	
30457	0	0	2K	259.7M	256.3M	0K	0K	7%	upload	
30926	0	0	2K	259.7M	256.3M	0K	0K	7%	upload	
30925	0	0	2K	259.7M	256.3M	0K	0K	7%	upload	
30456	0	0	2K	259.7M	256.3M	0K	0K	7%	upload	
30425	0	0	2K	259.7M	256.0M	0K	0K	7%	upload	
30455	0	0	2K	259.7M	252.3M	0K	0K	6%	upload	
29272	137	0	34499K	1.0G	103.8M	0K	-24K	3%	chrome	
3456	0	0	3K	1.1G	91232K	0K	0K	2%	simpres.s.bin	
4680	331	0	74K	640.4M	70424K	0K	892K	2%	firefox	
30574	0	0	34499K	958.3M	52416K	0K	0K	1%	chrome	
30549	41	0	34499K	622.4M	44972K	0K	0K	1%	chrome	
3343	0	0	127K	51148K	38872K	0K	0K	1%	atop	
29211	0	0	34499K	1.5G	30600K	0K	0K	1%	chrome	
1530	3181	0	1778K	157.7M	29240K	0K	0K	1%	Xorg	

The MEM line is displayed in cyan because the amount of memory that can quickly be claimed is small (free plus the page cache). The processes are not yet in the danger zone because no pages are swapped out (PAG line, `swout`). Better yet, `firefox` has physically referenced another 892KiB (that is a small amount compared to the 156MiB that `lekker` got).

We can see that `chrome` has shrunk by 24KiB (6 pages). We later found out that this was caused by malloc'ed memory pages that were freed, followed by a new malloc of 6 pages without referencing the pages again. Hence the virtual and resident size at first shrunk by 6 pages, after which only the virtual size grew by 6 pages. After all, the resident size shrunk without a change of the virtual size.

Case study: The kernel gets worried....

Four snapshots of 20 seconds later, we see that `lekker` has an unsatisfiable hunger: it has grown more than 600MiB (virtual and resident) since the previous screen shot, which is about 150MiB per 20 seconds.

ATOP - sammy		2010/03/02		15:13:52		----		20s elapsed		
PRC	sys	2.49s	user	7.14s	#proc	229	#zombie	0	#exit	0
CPU	sys	12%	user	35%	irq	0%	idle	147%	wait	6%
CPL	avg1	0.70	avg5	0.52	avg15	0.69	csw	49194	intr	38728
MEM	tot	3.8G	free	97.9M	cache	314.2M	buff	87.7M	slab	224.5M
SWP	tot	8.0G	free	7.8G			vmcom	4.1G	vmlim	9.9G
PAG	scan	30712	stall	0			swin	91	swout	322
LVM	vg00-lvswap		busy	1%	read	91	write	322	avio	0.59 ms
LVM	vg00-lvusr		busy	1%	read	9	write	0	avio	12.8 ms
LVM	vg00-lvhome		busy	1%	read	0	write	25	avio	4.32 ms
DSK	sda		busy	3%	read	41	write	60	avio	5.21 ms
NET	network		ipi	4	ipo	0	ipfrw	0	deliv	0
NET	wlan0	----	pcki	4	pcko	0	si	0 Kbps	so	0 Kbps

PID	MINFLT	MAJFLT	VSTEXT	VSIZE	RSIZE	VGROW	RGROW	MEM	CMD	1/17
31048	40020	0	2K	941.7M	938.3M	156.3M	156.3M	24%	lekker	
30457	0	0	2K	259.7M	256.3M	0K	0K	7%	upload	
30456	0	0	2K	259.7M	256.3M	0K	0K	7%	upload	
30925	0	0	2K	259.7M	256.2M	0K	-76K	7%	upload	
30425	0	1	2K	259.7M	256.0M	0K	-104K	7%	upload	
30926	7	4	2K	259.7M	255.6M	0K	-688K	7%	upload	
30455	0	0	2K	259.7M	252.3M	0K	-4K	6%	upload	
29272	119	0	34499K	1.0G	104.0M	0K	224K	3%	chrome	
3456	0	0	3K	1.1G	90840K	0K	-352K	2%	simpres.sbin	
4680	109	6	74K	640.4M	70188K	0K	-192K	2%	firefox	
30574	0	2	34499K	958.3M	52404K	0K	-12K	1%	chrome	
30549	42	0	34499K	622.4M	44964K	0K	-8K	1%	chrome	
3343	0	0	127K	51148K	38872K	0K	0K	1%	atop	
29211	0	0	34499K	1.5G	30520K	0K	-80K	1%	chrome	

The page cache has been shrunk as well as the slab (e.g. in-core inodes and directory entries). Processes weren't spared either. They didn't shrink virtually, but some of their resident pages were taken away by the kernel (negative `RGROW`). Because the worried (but not yet desperate) kernel is looking hard for memory to free, more and more pages are checked by the page scanner: 30712 pages were verified to see if they are candidate to be removed from memory (`PAG` line, `scan`). If a page has to be removed from memory that was modified, that page has to be saved to the swap space. In this snapshot, 322 pages were written to the swap disk (`PAG` line, `swout`). This resulted in 322 writes to the swap space logical volume (`vg00-lvswap`) that were combined to about 60 writes to the physical disk (`sda`). Because so many pages were swapped out in a short time, the `PAG` line is displayed in red.

However, processes don't sit still and some of the pages that were swapped out will be referenced again. These pages are read again which happened 91 times (`PAG` line, `swin`). Fortunately `atop` itself makes all its pages resident at startup and locks them in memory, thus preventing them to be swapped out and making the measurements unreliable.

Case study: The kernel gets desperate as well as the users...

The memory-leaking process `lekker` cannot be stopped. We fast-forward 5 minutes:

ATOP - sammy		2010/03/02 15:18:52		----		20s elapsed				
PRC	sys	2.54s	user	0.82s	#proc	229	#zombie	1	#exit	1
CPU	sys	12%	user	4%	irq	0%	idle	63%	wait	120%
CPL	avg1	12.77	avg5	6.18	avg15	2.88	csw	14932	intr	30144
MEM	tot	3.8G	free	29.6M	cache	41.9M	buff	0.7M	slab	95.6M
SWP	tot	8.0G	free	6.5G			vmcom	5.9G	vmlim	9.9G
PAG	scan	121298	stall	0			swin	8440	swout	32135
LVM	vg00-lvswap		busy	98%	read	8432	write	32135	avio	0.49 ms
LVM	vg00-lvhome		busy	92%	read	44	write	55	avio	188 ms
LVM	vg00-lvusr		busy	75%	read	267	write	0	avio	57.2 ms
DSK	sda		busy	98%	read	3972	write	428	avio	4.55 ms
NET	network		ipi	3	ipo	1	ipfrw	0	deliv	0
NET	wlan0	----	pcki	5	pcko	3	si	0 Kbps	so	0 Kbps

PID	MINFLT	MAJFLT	VSTEXT	VSIZE	RSIZE	VGROW	RGROW	MEM	CMD	1/17
31048	32016	0	2K	2.8G	2.0G	125.1M	34212K	52%	lekker	
30457	89	69	2K	259.7M	253.1M	0K	24K	6%	upload	
30456	588	261	2K	259.7M	247.4M	0K	3180K	6%	upload	
30455	0	0	2K	259.7M	243.8M	0K	-120K	6%	upload	
30925	149	231	2K	259.7M	184.8M	0K	-3744K	5%	upload	
30425	1979	411	2K	259.7M	138.5M	0K	6980K	4%	upload	
30926	224	254	2K	259.7M	90332K	0K	-18.9M	2%	upload	
29272	134	6	34499K	1.0G	86560K	0K	-656K	2%	chrome	
3343	0	0	127K	51148K	38872K	0K	0K	1%	atop	
4680	91	111	74K	640.4M	36500K	0K	-2904K	1%	firefox	
30574	0	0	34499K	960.7M	29576K	0K	-64K	1%	chrome	
3456	99	102	3K	1.1G	29252K	0K	-876K	1%	simpres.sbin	
30549	43	3	34499K	622.4M	18372K	0K	-44K	0%	chrome	
29211	144	25	34499K	1.5G	16824K	0K	420K	0%	chrome	

By now, `lekker` has grown to a virtual size of 2.8GiB of which 2GiB is resident (more than 50% of the physical memory of the system). In the past 20 seconds, `lekker` has tried to get hold of 128MiB more virtual memory but has only been able to make 34MiB resident. We know from the past that `lekker` tries to make all its virtual memory resident as soon as it can, so we can conclude that the kernel is very busy swapping out pages. The page cache has already been minimized, as well as the inode cache and directory entry cache (part of the slab). Obviously the processes will also have to “donate” physical memory. One of the `upload` processes (PID 30926) is even donating 19MiB. We can see for some of the `upload` processes that they had to give back quite a lot more physical memory (RSIZE). They had 6 times 256MiB, of which they now have only two thirds.

The system is swapping out heavily (32135 pages in the last 20 seconds) but is also swapping in (8440 pages). Because of this, the PAG line is displayed in red. The disk is very busy (the DSK as well as LVM lines are red). The average service times of requests for the logical volumes that are not related to swapping (`lvhome` and `lvusr`) are getting longer because the requests to those areas are swamped by requests to swap space (`lvswap`). Although a relatively small number of requests are related to `lvusr` and `lvhome`, these logical volumes are busy respectively 75% and 92% of the time. The system feels extremely slow now. Time to get rid of the leaking process.....

Case study: Relief...

Five minutes later, the big spender lekker has finished and thus not using memory any more:

ATOP - sammy		2010/03/02		15:23:52		----		20s elapsed		
PRC	sys	1.89s	user	2.53s	#proc	229	#zombie	1	#exit	1
CPU	sys	9%	user	12%	irq	1%	idle	68%	wait	110%
CPL	avg1	11.35	avg5	9.85	avg15	5.39	csw	21621	intr	27397
MEM	tot	3.8G	free	1.8G	cache	45.1M	buff	0.9M	slab	93.2M
SWP	tot	8.0G	free	7.2G			vmcom	3.4G	vmlim	9.9G
PAG	scan	32	stall	0			swin	7384	swout	0
LVM	vg00-lvswap	busy	98%	read	7384	write	0	avio	2.71 ms	
LVM	vg00-lvhome	busy	71%	read	85	write	77	avio	88.6 ms	
LVM	vg00-lvtmp	busy	36%	read	0	write	10	avio	738 ms	
DSK	sda	busy	98%	read	4119	write	40	avio	4.81 ms	
NET	transport		tcpi	18	tcpo	16	udpi	0	udpo	0
NET	network		ipi	22	ipo	18	ipfrw	0	deliv	18
NET	wlan0	----	pcki	23	pcko	19	si	0 Kbps	so	2 Kbps

PID	MINFLT	MAJFLT	VSTEXT	VSIZE	RSIZE	VGROW	RGROW	MEM	CMD	1/18
30457	0	0	2K	259.7M	253.0M	0K	0K	6%	upload	
30455	174	274	2K	259.7M	243.0M	0K	1788K	6%	upload	
31071	44	100	2K	259.7M	241.7M	0K	576K	6%	upload	
30456	15	8	2K	259.7M	238.0M	0K	92K	6%	upload	
30925	313	360	2K	259.7M	179.4M	0K	2692K	5%	upload	
30425	540	448	2K	259.7M	142.0M	0K	3952K	4%	upload	
30926	1019	455	2K	259.7M	103.2M	0K	5896K	3%	upload	
29272	126	3	34499K	1.0G	89364K	0K	-508K	2%	chrome	
3343	0	0	127K	51148K	38872K	0K	0K	1%	atop	
4680	286	323	74K	640.4M	33784K	0K	1804K	1%	firefox	
30574	67	58	34499K	960.7M	29200K	0K	492K	1%	chrome	
3456	395	227	3K	1.1G	27504K	0K	1312K	1%	simpres.s.bin	
30549	80	32	34499K	622.4M	18332K	0K	256K	0%	chrome	

However, the effect of lekker as a memory hog can be noticed for a long time. We can see that the upload processes are slowly referencing their swapped-out pages resulting in a resident growth again. Because there is an ocean of free space (1.8 GiB), nothing is swapped out any more and hardly any scanning (PAG line, scan).

We see a lot of major page faults (MAJFLT) for processes: references to virtual pages that are retrieved from disk. Either they were swapped out and now have to be swapped in, or they are read from the executable file. The minor page faults (MINFLT) are references to virtual pages that can be made resident without loading the page from disk: pages that need to be filled with zeroes (e.g. for malloc's) or pages that were "accidentally" still available in the free page pool.

The disk is still very busy retrieving the virtual pages that are referenced again and need to be swapped in (swin is 7384 which corresponds to read for logical volume lvswap). Therefore the DSK and some LVM lines are shown in red. The physical disk sda is mainly busy due to the requests of logical volume lvswap. However this also slows down the requests issued for the other logical volumes. One request to lvtmp even takes 738ms!

Case study: Life's almost good again...

More than seven minutes later, we can see that the system is almost tranquil again:

ATOP - sammy		2010/03/02		15:31:12		----		20s elapsed		
PRC	sys	4.66s	user	7.23s	#proc	228	#zombie	0	#exit	1
CPU	sys	23%	user	35%	irq	1%	idle	111%	wait	30%
CPL	avg1	1.91	avg5	4.49	avg15	4.57	csw	41583	intr	48545
MEM	tot	3.8G	free	996.5M	cache	157.0M	buff	240.4M	slab	141.9M
SWP	tot	8.0G	free	7.7G			vmcom	3.4G	vmlim	9.9G
PAG	scan	0	stall	0			swin	664	swout	0
LVM	vg00-lvusr		busy	44%	read	2903	write	0	avio	3.10 ms
LVM	vg00-lvswap		busy	2%	read	664	write	0	avio	0.69 ms
LVM	vg00-lvhome		busy	1%	read	0	write	49	avio	3.92 ms
DSK	sda		busy	47%	read	3094	write	40	avio	3.06 ms
NET	transport		tcpi	235	tcpo	172	udpi	9	udpo	9
NET	network		ipi	249	ipo	183	ipfrw	0	deliv	244
NET	wlan0	----	pcki	250	pcko	184	si	135 Kbps	so	8 Kbps

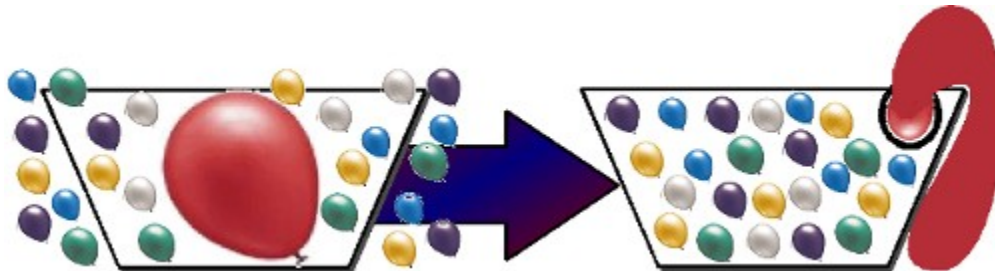
PID	MINFLT	MAJFLT	VSTEXT	VSIZE	RSIZE	VGROW	RGROW	MEM	CMD	1/18
30457	0	0	2K	259.7M	256.3M	0K	0K	7%	upload	
30456	0	0	2K	259.7M	255.9M	0K	0K	7%	upload	
30925	0	0	2K	259.7M	255.1M	0K	0K	6%	upload	
31071	0	0	2K	259.7M	255.0M	0K	0K	6%	upload	
30926	483	188	2K	259.7M	252.2M	0K	2684K	6%	upload	
30455	0	0	2K	259.7M	252.0M	0K	0K	6%	upload	
30425	0	0	2K	259.7M	247.3M	0K	0K	6%	upload	
29272	136	0	34499K	1.0G	95864K	0K	-200K	2%	chrome	
3456	0	0	3K	1.1G	80776K	0K	0K	2%	simpres.bin	
4680	3581	0	74K	767.4M	75628K	6144K	7152K	2%	firefox	
3343	0	0	127K	51148K	38872K	0K	0K	1%	atop	
30574	0	0	34499K	960.7M	34448K	0K	0K	1%	chrome	
29211	0	0	34499K	1.5G	27064K	0K	0K	1%	chrome	

There is far less disk I/O and certainly not all disk I/O is related to swapping any more. Processes (like upload) still do not have all their resident memory back, because they simply haven't touched all of their virtual pages since the "storm" has passed. Probably many of these pages have been used during their initialization phase and will not even be referenced any more. As such a small brease might help to clean up a dusty memory, however a stormy leaker as lekker can better be avoided.....

Possible solutions for memory leakage

From the case study, it is clear that only one misbehaving process can cause a heavy performance degradation for the entire system. The most obvious solution is to solve the memory leakage in the guilty program and take care that every malloc'ed area is sooner or later freed again. However, in practice this might not be a trivial task since the leaking program will often be part of a third-party application.

Suppose that a real solution is not possible (for the time being), it should be possible to avoid that the leaking process is bothering *other* processes. Preferably it should only harm its own performance by limiting the resident memory that the leaking process is allowed to consume. So not allowing the balloon to expand unlimited (pushing out the others), but putting a bowl around it redirecting the superfluous expansions outside the box....



The good news is: there is a standard `ulimit` value to limit the resident memory of a process.

```
$ ulimit -a
....
max memory size          (kbytes, -m) unlimited
```

The default value is “unlimited”. The command `ulimit` can be used to set a limit on the resident memory consumption of the shell and the processes started by this shell:

```
$ ulimit -m 409600
$ lekker &
```

The bad news however is: this method only works with kernel version 2.4, but not any more with kernel version 2.6 (dummy value).

But there is other good news (without related bad news this time):

In the current 2.6 kernels a new mechanism is introduced called *container groups* (cgroups). Via cgroups it is possible to partition a set of processes (threads) and specify certain resource limits for such partition (container). A cgroup can be created for all kind of resources, also for memory. It is beyond the scope of this document to go into detail about cgroups, but a small example can already illustrate the power of this mechanism.

Cgroups are implemented via a filesystem module, so first of all the virtual cgroup filesystem (with option “memory”) should be mounted to an arbitrary directory. This mount has to be done only once after boot, so it's better to specify it in your `/etc/fstab` file:

```
# mkdir /cgroups/memo
# mount -t cgroup -o memory none /cgroups/memo
```

To define a new memory cgroup for the leaking process(es):

1. Create a subdirectory below the mount point of the virtual cgroup filesystem:

```
# mkdir /cgroups/memo/leakers
```

At the moment that you create a subdirectory, it is magically filled with all kind of pseudo files and subdirectories that can be used to control the properties of this cgroup.

2. One of the “files” in the newly created subdirectory is called `memory.limit_in_bytes` and can be used to set the total memory limit for all processes that will run in this cgroup:

```
# echo 420M > /cgroup/memo/leakers/memory.limit_in_bytes
```

3. Another “file” in the newly created directory is called `tasks` and can be used to specify the id's of the processes/threads that must be part of the cgroup. If you assign a process to a cgroup, also its descendents (started from then on) will be part of that cgroup. Suppose that the leaking process `lekker` runs with PID 2627, it can be assigned to the cgroup `leakers` as follows:

```
# echo 2627 > /cgroup/memo/leakers/tasks
```

Now the leaking process can not use more resident memory than 420MiB. When it runs, `atop` might show the following output:

MEM	tot	3.8G	free	1.9G	cache	369.1M	buff	390.1M	slab	247.8M	
SWP	tot	8.0G	free	4.8G			vmcom	4.2G	vmlim	9.9G	
PAG	scan	0	stall	0			swin	0	swout	40913	
DSK		sda	busy	20%	read	29	write	708	avio	5.47 ms	
	PID	MINFLT	MAJFLT	VSTEXT	VSIZE	RSIZE	VGROW	RGROW	MEM	CMD	1/6
	2627	34017	0	2K	3.6G	374.7M	132.9M	-26.8M	10%	lekker	
	2532	0	0	3K	1.1G	119.2M	0K	0K	3%	simpres.bin	
	1702	0	0	1779K	453.5M	100.9M	0K	0K	3%	Xorg	
	1992	0	0	1764K	1.1G	23844K	0K	0K	1%	nautilus	

The line labeled MEM shows that 1.9GiB memory is free. On the other hand, the line labeled PAG shows that a lot of pages have been swapped out.

The process `lekker` has already grown to 3.6GiB virtual memory (VSIZE), but it only uses 374MiB resident memory (RSIZE). During the last sample, the process has even grown 132MiB virtually (VGROW), but it has shrunk 26MiB physically (RGROW). And what's more important, the other processes are not harmed any more by the leaking process. Their resident growth is not negative. The leakage is not fixed, though under control...

