Memory usage analysis

System-wide memory analysis

Top/Free

These two command-line tools are the two most well-known and (especially with the default columns shown) very useless and misleading.

Free

In free the used swap shows the amount of used swap and used -/+ buffers/cache is supposed to show the amount of really used RAM. The latter is computed as all memory used minus various caches and buffers, however since sizes reported for caches and buffers don't reflect reality very well (they include actually really used memory), the numbers are useless for anything but getting a very rough picture.

Top

The totals in top are the same case like with free. The values reported for processes that are shown by default are not of much use either - their meaning is unclear or they are of no practical use. It is possible to change the columns shown using the f key to get some more useful values. However top is still only useful for rough analysis as it doesn't provide any details and does not account at all for sharing memory between processes.

Explanation of some of the columns (some of which may be wrong though because of their confusing meaning and values):

* VIRT - the size of allocated address space. Not memory, but address space. This value in practice means next to nothing. When a process requests a large memory block from the system but uses only a small part of it, real usage will be low, VIRT will be high.
* RES - resident memory size, i.e. the memory that the process uses in RAM. While this value has some practical value it is not that useful because it includes all memory this process possibly shares with other processes (e.g. KDE applications share large KDE libraries and the size of these libraries is includes in RES of every such application). Swap is not included which additionally decreases the usefulness of this value.
* SHR - the amount of memory that could be potentially shared with other processes. The meaning is somewhat unclear and this value is of little practical value and it is not useful on its own.
* SWAP - the amount of swapped out memory - not only memory moved to the swap but also for example memory-mapped files, which again makes this value not very useful.
* CODE - memory taken by executable code. Various tables used by the code (e.g. virtual tables) don't seem to be included though - not very useful.

* DATA - it clearly does not mean what the manual page says and the actual meaning of the value is unclear.

See the manual page for top for more details. Note however that the manual page (as of procps version 2.3.7 and Linux version 2.6.18) is either obsolete or appears to be wrong in description of some of the fields (e.g. DATA).

/proc

Various files in the /proc filesystem can provide information about memory. However they're are poorly documented (linux/Documentation/filesystems/proc.txt) and many of them are broken (and since top and free get their values the same way, they are equally broken). In /proc/PID/ there are several files:

* status - human-readable data about the process, includes some data shown by top
* stat/statm - similar, not in human-readable format
* maps - lists memory mappings of the process (TODO if somebody feels like documenting, feel free to do so)
* smaps - lists memory mappings of the process with more details (TODO if somebody feels like documenting, feel free to do so)

Exmap

Exmap (http://www.berthels.co.uk/exmap/) is probably the best currently available tool for system-wide memory analysis. Advantages include detailed information about memory, accounting for shared memory and, last but not least, reporting values that make sense. Since exmap is a relatively new tool it's not in very wide-spread use, so you will possibly have to download and build it yourself (it requires gtkmm and boost libraries). At http://ktown.kde.org/~seli/download/exmap/ is my lousy attempt at SUSE packages (unsupported).

Exmap has documentation that explains its use and the meaning of the various values. In short:

* Run exmap. It may be better to run exmap as root in order to get access to all memory.
* The 'Processes' tab shows memory information about each process in the first listview. The values are:
    - VM - VIRT from top - not really useful.
    - Mapped size - the total of memory actually used by the process, both in RAM and swap. Note however that e.g. libraries are not swapped out to swap but simply discarded (and read again from the files if needed). This value includes even shared memory.
    - Resident size - mapped size, but only in RAM (without swap).

o Sole mapped - memory used only by this process. For example if a process uses a shared library that no other process uses at the moment it is included here.

o Writable - memory with the processes' private data. It is part of sole mapped that the process has already written to.

o Effective - the effective values are an attempt to compute how much memory a process uses in practice. Effective mapped/resident are mapped/resident values adjusted for shared memory. Memory that is shared by more processes is equally divided among them, i.e. if 10 processes use a 10MB large shared library (and each of them really uses the whole library), the library is counted as 10MB in mapped/resident values but only as 1MB in effective values.

* The second listview shows memory mappings for the selected process. There are memory-mapped files and binaries and also several special mappings:

o [heap] - dynamically allocated memory (i.e. malloc etc.)

o [stack] - the stack of the process

o [anon] - anonymous mapping from the mmap() system call - they should usually be viewed the same like [heap] although with multithreaded applications some [anon] mappings may be another [stack] mappings

* The remaining two listviews provide details about the mappings. For experts (see exmap documentation).

Exmap in practice: Run exmap (possibly as root, if you get messages about failures to open files and you need them). Sort processes by effective mapped size. Higher values are worse. Use the second listview to find out which file is possibly responsible or if the high memory usage comes from the process data ([heap], [anon] and [stack]). It usually cannot be changed which libraries are used, so the values that should be actually checked are writable and sole mapped columns - they are the memory actually used by the application itself (although a portion may come from calling library functions of course).

X resources analysis

Applications using the X window system allocate some resources (such as pixmaps and windows) in the X server process and refer to them only using their handles (ID numbers). The memory for these resources is allocated in the X server process. The xrestop tool shows resource usage of applications in a way similar to top (without being seriously broken). The two most important columns are 'Pxms mem', which is memory taken by pixmaps (QPixmap), and 'Other', which is memory taken by other resources (should be usually low, high value may indicate a leak in code using directly Xlib, possibly in some library). TODO: there should be a tool to help detect pixmap leaks

Application memory analysis

Intro

Applications use memory in several ways that can be seen in exmap. There is memory taken by binaries, there are data segments from binaries (such as global variables but also e.g. relocation tables) and there is dynamic memory used. Dynamic memory is stack and mainly heap. Heap is part of memory for allocations using malloc(), operator new and similar functions.

There are two ways memory is allocated in the heap, as far as memory usage is concerned. There is one main heap area (shown as [heap] in exmap) that has one end fixed and other end is moved using the brk() system call. So when an allocation needs to be done a part of this area is reserved (recorded in heap internal structures) and its address is returned. If there's no more contiguous free space available the area is enlarged by moving the upper end using the brk() call. This has several consequences:

* Every allocation has a certain overhead - when allocation many very small blocks the overhead caused by heap internal structures can significantly increase the actual memory usage. Various additional implementations of memory allocation like pools, arenas or obstacks can be used to reduce this problem (they use a special area of memory and the whole area must be freed at once, so there's no need to keep track of each allocation).
* Memory can get easily fragmented - allocating blocks of different sizes can lead to heap having holes that are not big enough for following allocations. Heap implementations try to group small sizes together to reduce this problem and again additional memory allocation implementations like obstacks can reduce this problem as well (they use special area of memory that is freed at once so it cannot fragment the heap).
* The main heap area is one contiguous memory area that cannot be split into smaller parts - when a lot of memory is allocated, then one permanent allocation is made and previous allocations are freed, the permanent allocation keeps the movable end fixed and prevents shrinking (TODO: does malloc actually ever use brk() to shrink?). This problem again can be reduced by using a special allocation implementation with its own memory area. Heap implementations also try to reduce this problem by allocating separate memory for large allocations (usually 128KB and larger) using mmap() system call instead of brk() - these show as [anon] in exmap.

Note that using additional memory allocation implementations is usually difficult with Qt/KDE as the developer cannot control how classes like QString, QList etc. allocate memory for their data.

Memprof

Memprof (http://www.gnome.org/projects/memprof/) tracks malloc() calls and can therefore be used for analysing heap usage. Application needs to be launched using memprof (pay attention avoid forking using --nofork or similar) and they should have debug information available to get full details. Top bar shows memory usage (yellow - used, blue - peak usage, red - leaks, after pressing the Leaks), number of allocations (at the moment, does not include already freed memory) and bytes allocated (heap overhead is not included). After pressing the Profile button it can be examined where the allocations come from.

Memprof in practice: Compile application with debug info, run it using memprof. Press the Profile button when you want memory analysis. In the listbox select the item with the maximum total (should be __libc_start_main or main), in the listview on the right there will be a tree of allocations from this function and the functions called from it. Open the tree, follow the high values, find problems.

kmtrace

Kmtrace (from kdesdk/kmtrace in KDE SVN) has similar purpose to memprof but with different usage and different way of presenting results (the output format is text file and the usage needs more manual intervention).

Installation (e.g. to /tmp/kmtrace):

```
svn co svn://anonsvn.kde.org/home/kde/branches/KDE/3.5/kdesdk -N
cd kdesdk
svn co svn://anonsvn.kde.org/home/kde/branches/KDE/3.5/kdesdk/kmtrace
svn co svn://anonsvn.kde.org/home/kde/branches/KDE/3.5/kde-common/admin
make -f Makefile.cvs
./configure --prefix=/tmp/kmtrace
make && make install
```

Usage:

```
LD_PRELOAD=/tmp/kmtrace/lib/libktrace.so MALLOC_TRACE=kmtrace.out xterm
/tmp/kmtrace/bin/kmtrace kmtrace.out --tree kmtrace.tree >kmtrace.txt
```

Starting and finishing of tracking is done using functions ktrace() and kuntrace() from libktrace.so, so it is possible to finish tracking even before the application exits by calling kuntrace() manually (TODO kmtrace could be perhaps hacked to react on a signal or something):

```
gdb
attach [pid]
call kutrace()
quit
```

File kmtrace.out is an internal file that needs to be post-processed and can be deleted afterwards. File kmtrace.txt contains summary information and all allocations sorted by size - they're actually referred to as leaks, since tracking usually is finished after program exit, but when stopping tracking while the application is still running they represent all currently existing allocations. File kmtrace.tree provides all the allocations in a tree (first number is number of bytes, second is number of allocations). It is possible to limit the depth of the tree and to ignore subtrees with small sizes, see 'kmtrace --help'.

Valgrind

Valgrind's skin Massif ('valgrind --tool=massif application') traces memory usage of the application. After it exits it generates a .ps file showing allocations progress for main places and a text file with details (TODO: the text file does not seem very usable).

Valgrind can also detect memory leaks ('valgrind --tool=memcheck --leak-check=yes application', additionally also '--show-reachable=yes' may be used to show memory still allocated at application exit even if it's not unreachable). After application exit valgrind will print out backtrace of all leaked allocations (use '--num-callers=50' to get deeper backtraces).
HOWTO (AKA right to the point)

If you want to analyse memory usage of the whole system or to thoroughly analyse memory usage of one application (not just its heap usage), use exmap. For whole system analysis, find processes with the highest effective usage, they take the most memory in practice, find processes with the highest writable usage, they create the most data (and therefore possibly leak or are very ineffective in their data usage). Select such application and analyse its mappings in the second listview. See exmap section for more details. Also use xrestop to check high usage of X resources, especially if the process of the X server takes a lot of memory. See xrestop section for details.

If you want to detect leaks, use valgrind or possibly kmtrace (TODO memprof doesn't work for leaks for me). See their sections for more details.

If you want to analyse heap (malloc etc.) usage of an application, either run it in memprof or with kmtrace, profile the application and search the function call tree for biggest allocations. See their sections for more details.