1 Possible Solutions

Since exhausting the available primary memory is the problem, it is clear that a solution need to include using files on the hard drive to replace or extend the primary memory. We looked at the two most promising solutions and evaluated them in light of this specific problem.

1.1 Database

Description

A database could be used to store information about the packets, this would shift the responsibility of managing memory over to a database system. Possible candidates for use in this project are SQLite[1] and MySQL Embedded[2], both of which are supported on all platforms Wireshark runs on.

Evaluation

Database management systems (DBMS) are made for storing large amounts of data in a reliable way and make it possible to retrieve the data as fast as possible. DBMSes usually store their data on a hard drive, but use primary memory for caching to get higher throughput.

Using a database would mean that Wireshark would not itself need to manage its packet representation, but there are several drawbacks to this approach. First and foremost, a DBMS is designed with reliability and data-consistency as their primary goals and speed as a secondary goal. This has several implications, the most severe being that data would be written to disk as soon as it is added to the database. This would lead to slow performance compared to directly using memory even when there is available memory.

1.2 Memory-mapped Files

Description

An alternative for getting more memory than the machine's RAM is to use memory-mapped files. This works by creating files and mapping regions of these files to the virtual memory address space. After this is done, these files will be treated as they were part of the virtual memory, just like physical memory and paging files. The amount of memory that can be mapped in this way is limited by the amount of memory that can be addressed by CPU. On a 32-bit machine this limit is 2^{32} bytes = 4 GiB (even though this can be extended a bit by using PAE – this is not considered here). However, on a 64-bit machine this limitation currently not a problem as the amount of memory it can address is in the exabytes (10^{18}).

[1] `http://www.sqlite.org/`
[2] `http://www.mysql.com/products/embedded/`

Evaluation

Du to the limitations of 32-bits architectures mentioned above a solution with memory-mapped files will only be a real benefit on a 64-bit architecture. To be able to use more than 4 GiB of memory on a 32-bit computer one would have to mange memory oneself – unmapping (sections of) files to make room for new data when we run out of addressing space. This kind of memory management is difficult to do right and is very error-prone. Writing a memory manager as part of our modification is out of this project's scope.

There is also some issues with Windows compatibility. The mmap system call is part of the POSIX standard, but is not supported on Windows. There are equivalent system calls, though; VirtualAlloc/VirtualFree will accomplish the same on Windows, but this still means that we need to write Windows-specific code for this functionality.

Another issue to be aware of is the operating system's limitation of a process' virtual memory. On modern 64-bits OSes this is no longer a problem. 64-bit versions of Windows support up 8 TiB of virtual address space per 64-bit process [?], 64-bit Mac OS X's limit is 18 EiB [?], 64-bit Linux sets the limit to 128 TiB [?]. These limits are several orders of magnitude over what will be needed in the near future.

2 Chosen Solution

The group chose to base their solution on memory-mapped files for the reason outlined above. This solution will work on a 32-bit plattform, but it will be limited to 4 GiB of memory due to the mentioned reason. This means that this solution will only be of great help to users of 64-bit systems. The group considers this solution to be the best solution within the frame of this project.

2.1 Proof of concept

To confirm that our idea is feasible and to get familiar with the use of `mmap` to map a file into memory we wrote a simple test program in C – the same language Wireshark is written in. The program was tested on a 32-bit machine with 1 GiB of memory and used `mmap` to map two files, each with 1 GiB of zeroes, into the memory space and then read every thousand `double word` (4 bytes) and printed its address. The output from the program is shown in figure 1.

As the output shows, the mapped files span a memory area of $0xb7dc4000 - 0x37dc4000 = 2 \cdot 2^{30}$ bytes, i.e., 2 GiB. This was done with paging turned off, meaning that the system used 2 GiB of continouos address space with only one GiB of physical memory available, thus showing that it is possible for a process to use more memory than the sum of physical memory and operating system paging files.

```
$ ./mmap_test
37dc4000: 0
37dc4fa0: 0
37dc5f40: 0

[...]

b7dc19a0: 0
b7dc2940: 0
b7dc38e0: 0
```

Figure 1: Output from running the memory-mapping test program. The memory addresses span $2 \cdot 2^{30}$ bytes (2 GiB)