Τμήμα Επιστήμης Υπολογιστών

Performance Evaluation Of Kernel-Level Allocators In ARM And x86 Multicore Systems

Garcarz Aleksander

Intro

The kernel memory allocator is one of the most important parts of an operating system. It is responsible for managing memory by satisfying all the system's (kernel space) memory requests in the best way possible. (To avoid confusion) memory management is separated in two categories. The first category is memory management in kernel space and refers to the requests for all the memory needed to be allocated or freed by any process of the operating system. The other category is memory management in user space and refers to the memory requests from user space (application processes). Kernel memory allocation is placed in lower level than the user memory allocation.

Memory handling is a very sensitive process especially inside the kernel space since bad memory management can cause big memory waste and even more a great reduction of the general system's performance. The biggest issue for the memory of any system is memory fragmentation. Memory fragmentation is usually caused by allocating memory and freeing a part of it creating smaller non contiguous parts of the available memory. This way a big allocation could fail to be satisfied even though there is enough memory, because this memory is non contiguous. Memory fragmentation can also be caused by big allocations of memory in which only some objects remain allocated and the rest of the allocated memory is unused but cannot be returned to the operating system until all the objects are freed. When memory fragmentation becomes high and the most big allocations fail, a daemon process could be called to reduce the fragmentation by reallocating the active objects of the memory. But this process is time expensive and complex and a frequent use of it could affect the performance of a system a lot. Also, a big issue for the memory management is the time needed to satisfy a memory request. All the memory requests should be satisfied as fast as possible but usually fast allocations lead to big fragmentation due to the lack of examining the most efficient way to make the allocation without creating fragmentation. The most efficient way to solve the above memory issues is by kernel memory allocators making them efficient in terms of time and fragmentation. The most used kernel memory allocators are the slob allocator, slab allocator and slub allocator, all explained below.

The purpose of this work is to evaluate and understand how the implementation of each allocator affects the performance of the system, testing each using the appropriate workloads and recording stats about the execution. The performance of the allocators was recorded using memory and file i/o benchmarks on a x86 system and an arm64 system (for more details go to the methodology and the evaluation systems paragraph). Also, comparing the stats recorded on a x86 system and an arm64 system let us detect differences in the behaviour of the allocators on this two systems.

The results showed that in general the differences in behaviour between the used allocators is similar on both evaluated systems with some exceptions, explained below. Also as the results

showed there is no best allocator between the three allocators studied, but rather a trade off in performance for each one of them.

Kernel Memory Allocators

SLOB

Simple List Of Blocks (slob) allocator is the first of the linux kernel memory allocators. It is much simpler than the other allocators and needs small amount of memory for its implementation and functionality. In order to satisfy memory requests and keep track of the used memory SLOB maintains three lists of pages one for small objects (less than 256B) one for medium objects (less than 1024B) and one for big objects (bigger than 1024B). Allocation requests bigger than the page size are satisfied directly from the page level allocator. Each page consists of a list of free blocks. When a memory request occurs the allocator searches for a sufficient page to satisfy the request in the appropriate list of pages depending of the requested memory size. The blocks returned are found by the first fit algorithm. Although SLOB is simple, it faces a big issue. Inside each page many objects of different size are stored which probably can increase the fragmentation of memory. As described above SLOB is a very useful allocator especially for use in systems with limited memory, like embedded systems.

SLAB

SLAB allocator which has taken its name from one of its main functional units (slabs) is one of the most used allocators. In general SLAB allocator caches objects and maintaining their structure in an initialized state between uses. Before the description of the allocator it is important to understand its two main functional units caches and slabs. A cache in general is a piece of memory which can execute fast data accesses. Data is temporary stored in caches so future accesses can be executed faster. Similarly, in SLAB allocator caches are responsible to store objects for future uses. This approach absorbs the time taken by allocating initializing and freeing an object which in many cases creates a big time overhead. The other main unit are the slabs. A slab is a piece of memory and consists of one or more physically contiguous pages. At the beginning of a slab there is a small chunk of metadata containing information about the slab. Each slab is split into chunks, each of the size of a certain object. Slabs are separated in three different states: full, partial and free. Full slabs contain only used objects and can't satisfy any object requests. Partial slabs consist of used and free objects ready for reuse. Free slabs contain only free objects. The SLAB allocator maintains a doubly linked list of caches each for a certain

object type. Each cache contains three lists one for free slabs one for partial slabs and one for full slabs. The layout of slab allocator is shown in the figure below, from Goorman's book "Understanding the Linux virtual memory manager".

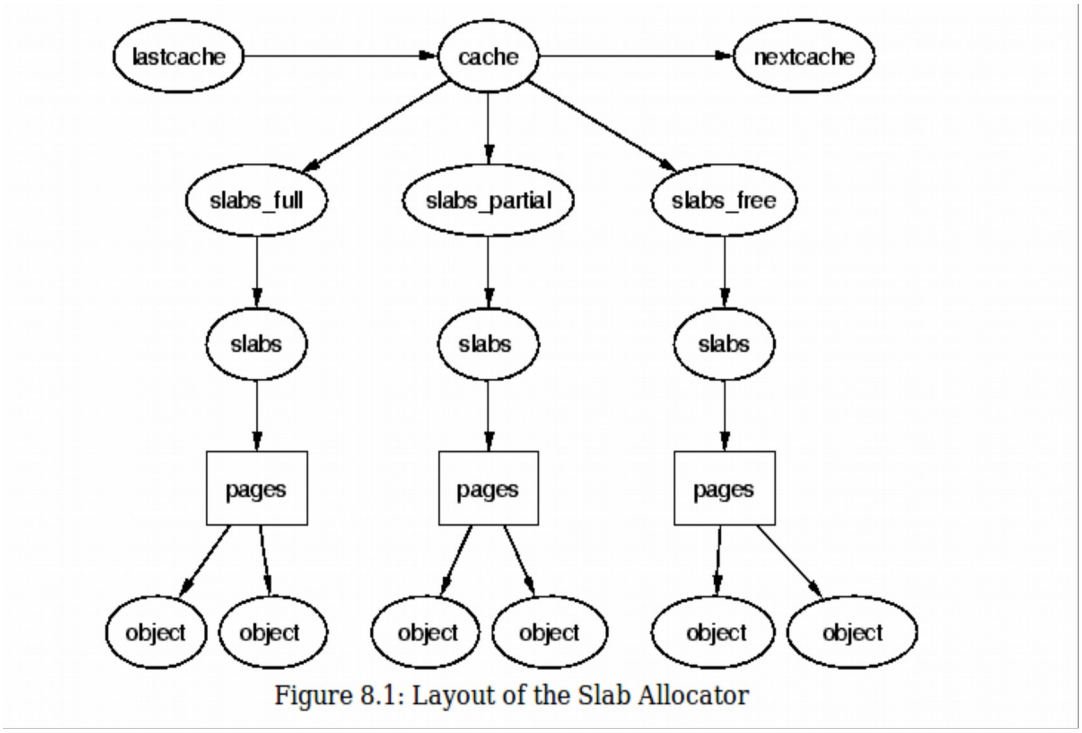

Figure 8.1: Layout of the Slab Allocator

This approach seems to be more effective in terms of time and fragmentation. Using caches time spent on allocating objects is eliminated since there is no need to allocate and initialize an object and instead just use one of the objects of a partial slab of the appropriate cache. Also fragmentation is reduced, since there is a cache for each type of object. Operating of slabs and caches are quite simple too. When a cache needs to grow it just adds a new slab and when it needs to return memory to the system just return a free slab. The metadata at the beginning of a slab contain a counter with the number of used objects (chunks) making easy to keep track when a slab becomes partial, free or full. When chunks become free or used only the counter and the linked list must be updated. However the SLAB allocator faces some important issues. Specifically the SLAB allocator maintains big amount of object queues per Node, per CPU. In large systems with many nodes and CPUs the storage overhead is big and the complexity of the management of those queues grows a lot. Furthermore the metadata at the beginning of each slab makes the alignment of the slab's objects harder.

SLUB

SLUB (unqueued slab allocator) was developed to improve some of the SLAB issues. SLUB

similarly to SLAB maintains a list of caches. The main difference is that each cache maintains only a list with partial slabs. When a partial slab becomes free is returned directly to the buddy allocator and when a slab becomes full it is ignored until it becomes partial again. This approach reduces the complexity of the allocator. Additionally instead of maintaining partial lists per node the SLUB allocator has a global pool of partial slabs. Also the metadata at the beginning of each slab was removed and placed in system memory map in the struct page representing each page of the system. This way, the alignment of the objects in each slab becomes easier. Also to improve scalability SLUB maintains a list of slabs per NUMA node.

An explanation of used tools and a basic background

Linux memory organization

Linux memory is separated in nodes. A node is a segment of memory which has a specific distance from the processor and as a result each memory access the processor attempts takes different time depending on how big this distance is. Hence separating memory in nodes Linux is able to reduce the memory access time making allocations from nodes which have the smallest distance from the CPU making this allocation (non uniform memory access). Also the memory is separated in zones each one responsible for satisfying different types of memory allocations. And more specifically each node is separated into three different zones depending on what system do we use. The zones are: Zone DMA which refers to the first 16MB of the system memory, Zone DMA32 which exists only in 64bit systems and refers to the memory from 16MB to 4GB, Zone Normal which refers to the rest memory from 4GB in 64bit systems and the memory from 16MB to 896MB in 32bit systems and zone HighMem which exists only in 32bit systems and refers to the rest of the memory from 896MB to 4GB.

Linux Memory Structs

Linux kernel represents each node using the `pglist_data` struct. All memory nodes are stored in a simply-linked `pglist_data` list and can be accessed by crossing this list using the `pglist_data *node_next` field of the struct. Except the `*node_next` field, `pglist_data` struct contains other necessary fields for memory management and one of them is the `zone_t node_zones` array containing `MAX_NR_ZONES` entries, one for each zone the current node consists of (usually[1] `MAX_NR_ZONES` is defined as 3). Hence each zone is described by the `zone_t` struct which contains many fields one of them is a `pglist_data *zone_pgdat` which is a pointer to

1 The number of the zones depends on the system's available memory and architecture. For example in a 64-bit system with 2 GB memory there will be only 2 zones: zone DMA and zone DMA32. In this case zone Normal which refers to the memory from 4GB and further is absent.

the node which this zone belongs to. Also the zone_t struct contains a free_area_t free_area array which is an array with MAX_ORDER entries (MAX_ORDER is defined as 11). The free_area array is used by the buddy allocator and each entry contains the list_head to a simply-linked list of free blocks of a certain size (depending on which entry is accessed). For example the first entry of the array contains the list head of a list which consists of zone's free blocks with size of 2^0*page_size, the second entry contains the list head of a list which consists of zone's free blocks with size of 2^1*page_size, etc.

Linux Proc File System

The proc/ file system is in general a virtual file system and is responsible for providing information about the system from kernel space where the user processes cannot access, to the user space. In most cases the data proc files are read-only but some kernel variables can be changed by the proc file system. Usually the size of the proc files is 0 bytes even though the files contain a lot of information which is updated each time a proc file is accessed. This becomes possible by the way linux manages files using the virtual file system which gives the flexibility of using different types of files like regular files or virtual files like proc files. That way when a user accesses a proc file the proc file system is triggered and creates the content.

Buddyinfo

Buddyinfo is a proc entry which helps visualise the free memory fragments of each zone in linux systems. Specifically buddy info accesses the pglist_data struct containing information for each node in memory and then reads the (zone_t) node_zones array representing all nodes' memory zones. Finally for each zone buddyinfo prints the number of free "chunks" of memory for each (free_area_t) free_area table entry representing different sizes of memory.

For example lets take a random buddy info output.

Node 0, zone	DMA	1	1	1	0	2	1	1	0	1	1	3
Node 0, zone	DMA32	99	36	27	29	10	9	5	1	3	3	792
Node 0, zone	Normal	250	521	297	137	149	90	39	19	9	11	2031

Assuming that the page_size is 4KB the output means that in this particular moment for zone DMA there are 1 chunk of 2^0*page_size (1 chunk of 4KB), 1 chunk of 2^1*page_size (1 chunk of 8KB)...and 3 chunks of 2^10*page_size (3 chunks of 4MB). Buddy info provides important information about the actual free fragments of memory and thus the total free memory of the system but on the other hand cannot provide information about the fragmentation recovery events happening.

Sysbench

Sysbench is a system benchmarking tool which provides many different benchmarking modes for

linux systems. Some of them are memory mode, file I/O mode, CPU mode, threads mode and more. For the evaluation of the performance of the three allocators studied the memory and the file I/O modes were chosen in order to study the behaviour of the allocators under memory and I/O pressure. When using the memory mode which tests the memory performance sysbench creates a buffer and perform random reads and writes of size of 32 or 64 bits until the total size of the buffer is red or written. Then the above process is repeated until the requested size of memory is transferred. When the workload is finished, sysbench displays the total time of the execution of the workload, the number of the performed operations, the number of the transferred memory and more. The file I/O mode consists of three phases: During the first phase, sysbench creates a number of test files that will be used for the execution of the workload. When the files are created the second phase starts. During this phase sysbench randomly perform reads and writes of a certain size in random files until the number of the transferred memory is reached. Then the third phase starts, which cleans all the created files for the workload. Similarly to the memory mode when the workload of file I/O is over sysbench displays information of the execution of the workload.

Methodology

For the execution of the evaluation three kernel images were created. Each of these images had enabled one allocator so there was created one image with slob allocator enabled one image with slab allocator enabled and one with slub allocator enabled. The enabled allocator was the only difference between the used images therefore no other modifications or options have been made or changed. To make the execution of the workloads and the record of the performance possible two bash scripts were created, one to monitor the performance of the allocator under memory operations pressure and one with disk operations pressure. The evaluation of the performance focuses on the execution time of the workload and at the same time on the memory consumption and fragmentation caused by each allocator.

To execute the workloads the monitoring and benchmarking tool sysbench was used. In order to create a big number of memory operations sysbench memory mode was used. The chosen amount of memory to be transferred was 150G and the number of threads for the execution was 8. The stats in the output of sysbench were used to record useful information like the time taken to make all the memory transfers and the memory bandwidth achieved during the workload. Memory consumption was computed using the proc entry buddyinfo comparing its output before and right after the execution of the workload. Also comparing the outputs return by the buddy info can show the situation of the available fragments of the free memory. For disk operations sysbench file i/o mode was executed. The number of files in which the random reads and writes would happen was 128. The number of threads for the execution was again 8 and the amount of memory to be transferred was 156MB.

To sum up the executed memory script at the beginning is calling the buddyinfo proc entry and store the output in a monitoring file. Then executes the memory workload using sysbench which monitors the stats of the execution (time and bandwidth). When the workload is finished monitors again the buddy info output making possible to detect any changes at the fragments of the memory. The file script starts with the creation of the files for the workload. When the files are ready the buddyinfo is called and the output is monitored in a file. Then the workload is executed using sysbench. Right after the execution another output of the buddyinfo is stored to the file and the script stops.

Each script was executed five times on each image and the results were summed up to avoid and absorb any possible deviation of the stats recorded. Each system used for the evaluation was rebooted between the execution of each script to avoid the alteration of the systems memory situation by the previous executions and minimize any noise created by other systems functions. In total five executions of the memory and five executions of the file i/o workload were recorded on each image. So the stats of the evaluation below were collected and summed up from 30 executions of the workloads on arm64 system and 30 on x86 system. The results are showed below.

Evaluation Systems

For the evaluation of the allocators' performance two systems were used: a x86 system and an arm64 system. The x86 system consists of an Intel Core i7-3770 CPU up to 3.40GHz with 4 cores and 8 threads, three levels of caches with 256KB, 1MB and 8MB capacity, 12GB RAM and a 500GB HDD (ATA Disk WDC WD5000AAKX-7). The arm64 system consists of a quad-core ARM Cortex-A53 MPCore up to 1.5 GHz, L1 I/D cache per core with 32KB capacity and a L2 cache with 1 MB capacity, a 2GB RAM and a SSD. The architecture and the specifications of the arm64 system are analysed in the link below:

https://www.xilinx.com/support/documentation/selection-guides/zynq-ultrascale-plus-product-selection-guide.pdf (see pages 5 and 6).

Results

This section shows the results of the evaluation for time and memory consumption, first for the arm64 system and then for the x86 system. After that, diagrams containing the memory fragments follow.

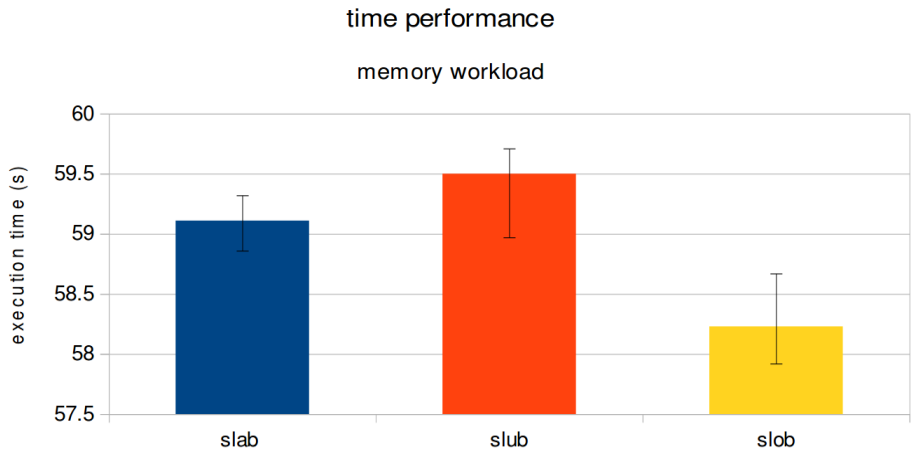

Figure 1a: arm64 system average time performance in memory workload.

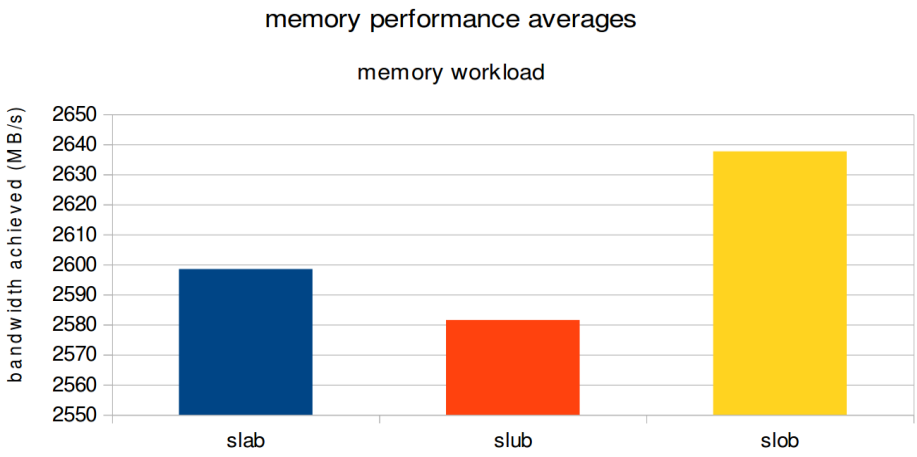

Figure 1b: arm64 system average bandwidth performance in memory workload.

Figure 1 shows the average time and bandwidth values of each allocator as calculated from all the executions of the memory workload. The two diagrams are inversely proportional since as the time taken for the execution grows the bandwidth decreases. Its clear that slob overtakes the performances of the other two allocators taking a lot less time to complete the benchmarks and as a result achieves the biggest bandwidth while transferring data. Second places the slab allocator followed by the slub with a difference of about 0,4 seconds in terms of time and 38 MB/s in terms of bandwidth. Additionally figure 1a contains an error window showing the best

and the worst performance achieved by each allocator during the execution of the workload. This error window is important since examining it makes it easy to estimate how strong the differences of the allocators' average performances are. During memory workload, slob's worst performance is clearly better than the best performance of the other two allocators. That means that slob achieved the best performance in all executions of the memory workload. At the same time slab's worst performance is worse than slub's best performance. In this case the slab allocator achieved better performance than slub in average but if the execution of the workloads was repeated, it could lead to different results. However recording the performance of the allocators five times for each benchmark and summing up the results makes the differences between the performances of the allocators strong enough to study their average performance and behavior.

Figure 2a: arm64 system average time performance in file i/o workload

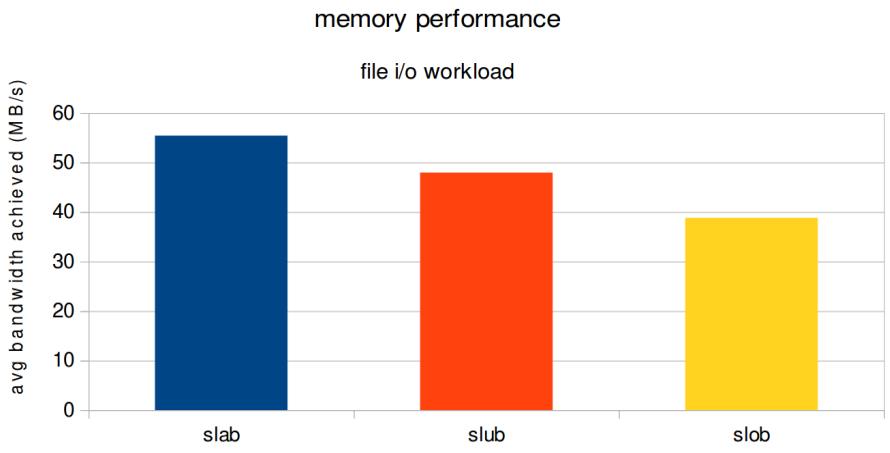

Figure 2b: arm64 system average bandwidth performance in file i/o workload

Figure 2 shows the average time and bandwidth performance achieved during file i/o workload. During this workload as seen on the diagram the slab allocator needs about 3 seconds in average while slub which follows needs almost 3,4 and slob over 4. The bandwidth diagram as expected shows that slab achieved the highest bandwidth and slob the lowest with the slub allocator to be in the middle. The offset between the best and the worst performance is the biggest using the slub allocator and the smallest using the slob allocator. Thus the performance of the slob allocator seems to be the most steady between the three allocators.

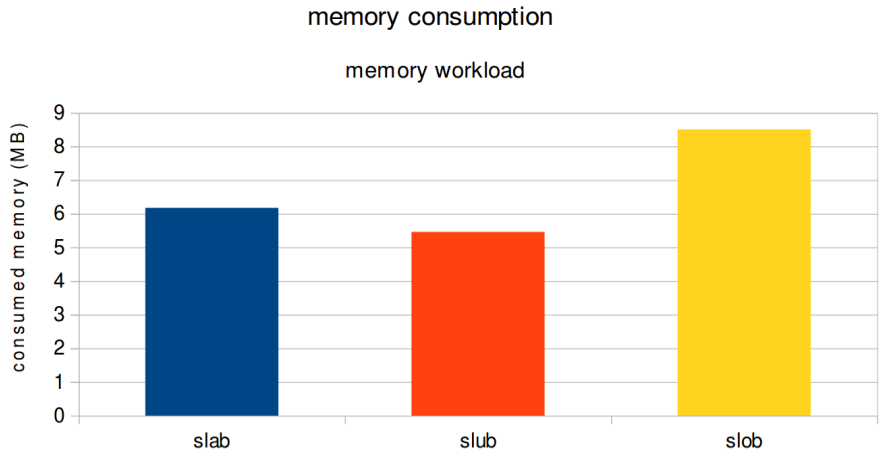

Figure 3a: arm64 memory consumption in memory workload

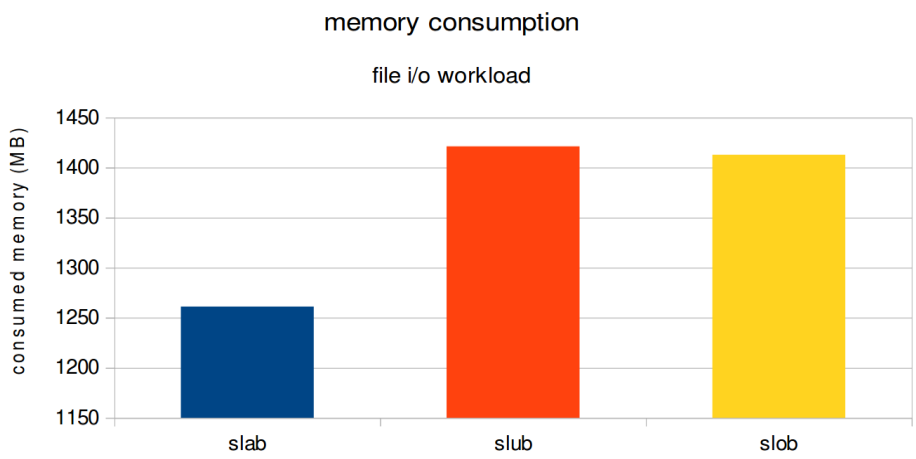

Figure 3b: arm64 memory consumption in file i/o workload

Figure 3 shows the average memory consumed during the execution of the workloads. Figure 3a shows the difference of the existing total free memory fragments before and right after the execution of the memory workload while figure 3b shows the difference of the existing free fragments of the memory before and right after the file i/o workload. As the first diagram shows the amount of consumed memory seems to be less with the slub allocator. The slob allocator causes the biggest consumption while the slab allocator is in the middle. In file i/o benchmark the slab allocator is responsible for the lowest memory consumption followed by the slob allocator and slub allocator. The difference of the consumed memory caused by the slub and the slob allocator is not that big comparing to the consumption caused by the slab allocator.

Now the evaluation diagrams of a x86 system follow. The results are calculated with the same way as the previous using the same workloads. Some differences between the performance of the arm64 and x86 system may be caused by the different hardware components (e.g. different size of ram) so the comparison between the two systems will take place comparing the behaviour of each allocator in relation to the others on each system. But firstly lets see the x86 diagrams.

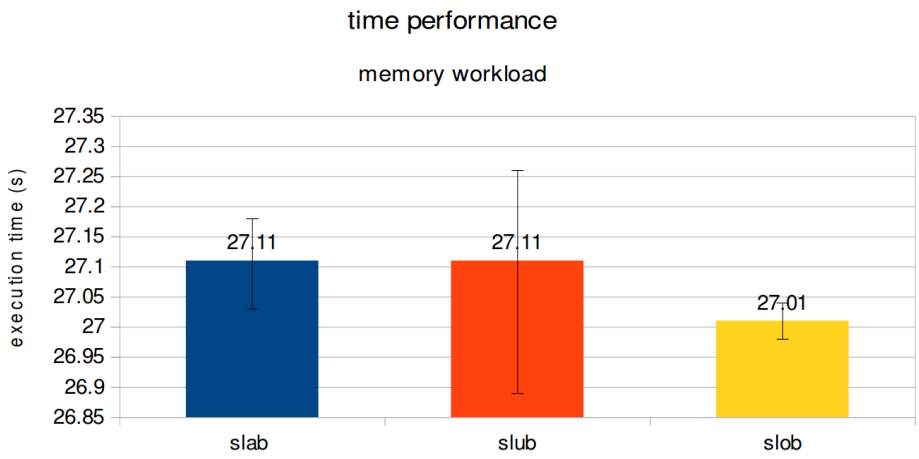

Figure 4a: x86 system average time performance in memory workload.

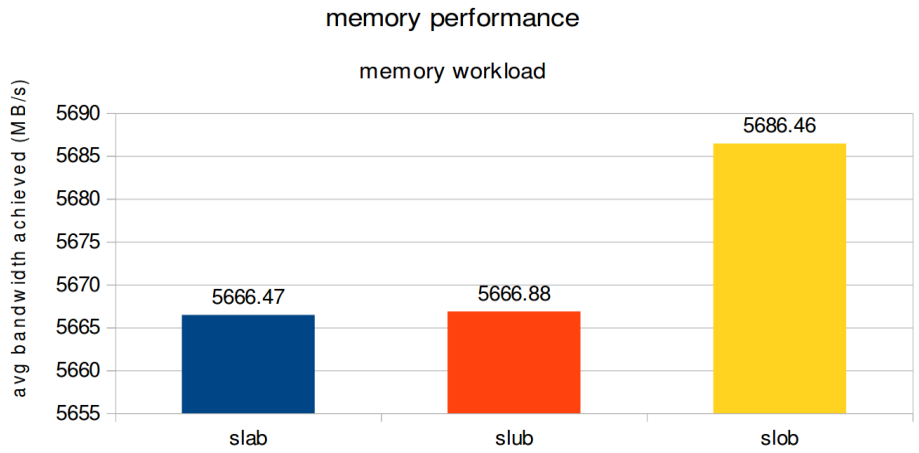

Figure 4b: x86 system average bandwidth performance in memory workload.

Figure 4 shows the average performance of each allocator during the memory workload. Its obvious that the slab and the slub executed the memory workloads in almost the same time in average achieving almost the same bandwidth. To be more accurate slab tends to be a bit worse than the slub since its average execution time was 27.10694 seconds with a 5666.468 MB/s bandwidth while slub's was 27.10562 seconds with a 5666.876 MB/s bandwidth. Although the

difference is so small that in this case the performances of the two allocators should be considered the same. Slob was the best allocator in the execution of the memory workload overtaking the other two allocators time by 0,1 s and almost 20 MB/s bandwidth. As figure 4a shows the differences between the times needed for the executions of memory workload are very small. The offset between the best and the worst performance is the biggest using slub allocator and the smallest using the slob allocator. Its noticeable that one of the slub's performances is the best and one is the worst between the performances of all the allocators.

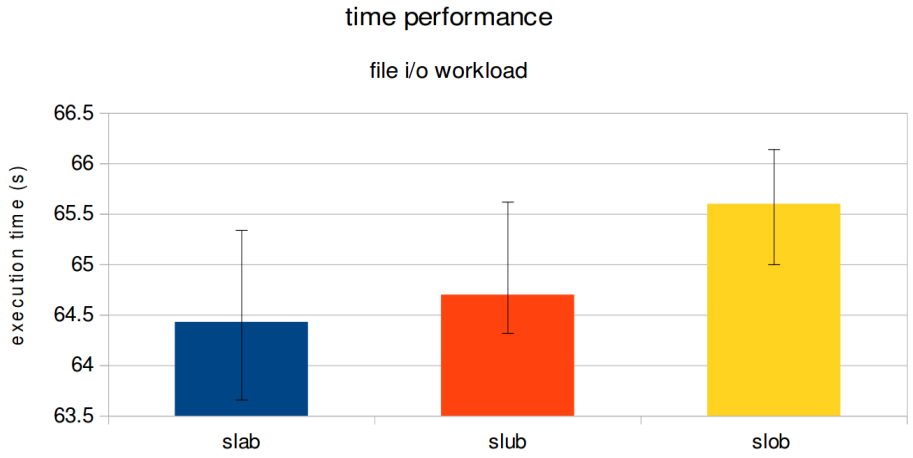

Figure 5a: x86 system average time performance in file i/o workload.

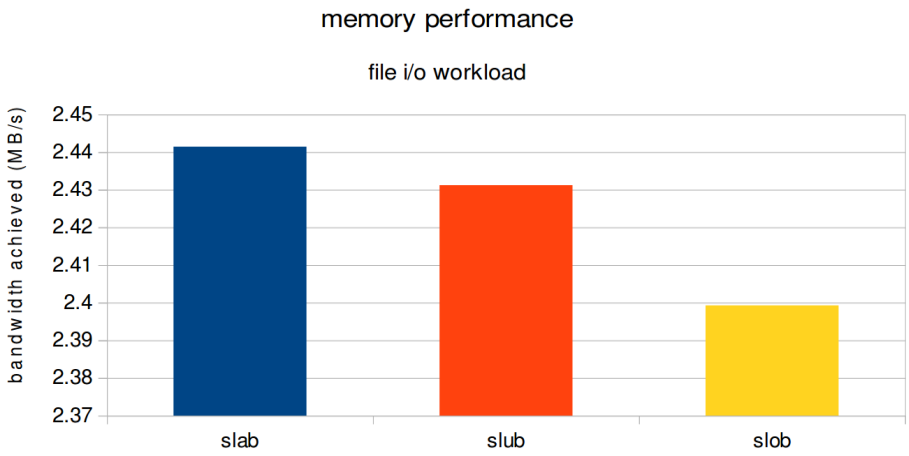

Figure 5b: x86 system average bandwidth performance in file i/o workload.

Figure 5 shows the average performance of each allocator under file i/o pressure. Looking at the diagrams its clear that slab is the one with the less time taken for the execution and therefore the one with the highest bandwidth. Slub follows with slightly worse performance with the time offset of slab's time reaching almost 0,2 second and 0,01 MB/s bandwidth. The allocator which seems to achieve the worst time and bandwidth performance is the slob allocator with an offset of almost 1 sec from the second slub in terms of time of the execution and 0,3 MB/s in terms of achieved bandwidth. The offsets between the best and the worst performance of the allocators during the execution of the file i/o seem to be similar, with the error window of the slab allocator to be the biggest one.

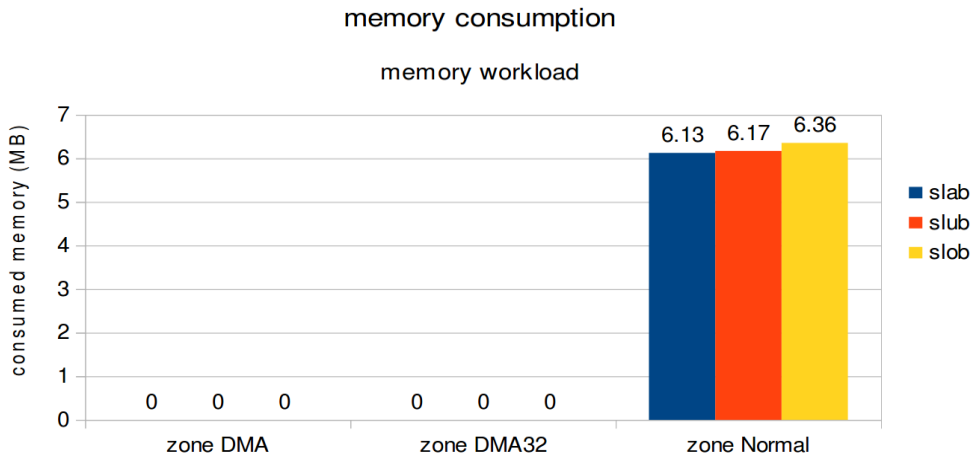

Figure 6a: x86 memory consumption in memory workload

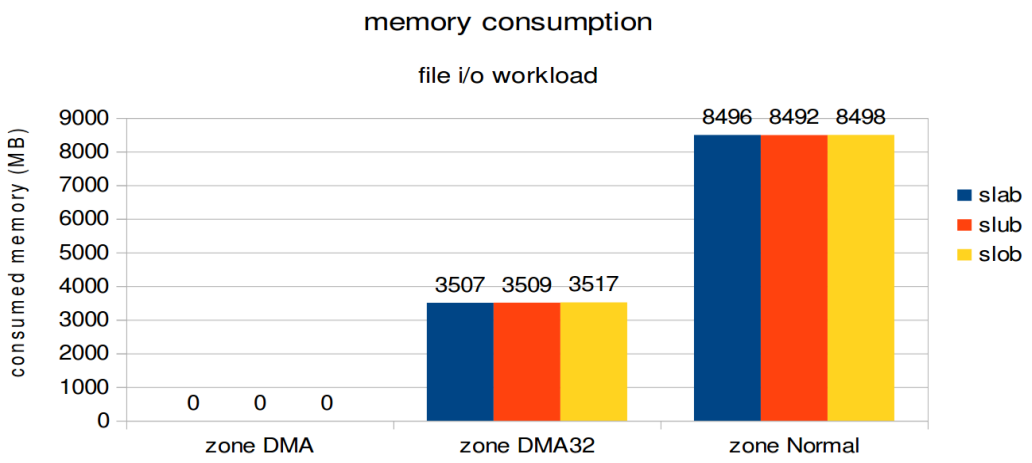

Figure 6b: x86 memory consumption in file i/o workload

After the diagrams of the time and bandwidth performance lets see how the execution of the memory and file i/o workload affected the memory of the x86 system (figure 6). The first diagram which shows the difference between the available free fragments of the memory after the memory benchmark shows that the slab allocator consumed less memory than the other allocators. Slob is the allocator with the biggest amount of consumed memory while slub's consumption is in the middle. The memory consumption during this workload affects only the zone Normal of the system keeping the zone DMA and zone DMA32 unaffected. On the other hand during the file i/o benchmark zone Normal is affected again but also zone DMA32 seems to have consumed memory. Specifically in zone DMA32 slab consumed the smallest amount of memory while the slob allocator consumed the biggest amount. In zone Normal slub allocator is the one with the less consumption followed by the slab with the slob's consumption to be the greatest. Finally, zone DMA seems to be unaffected like in memory workload.

A small evaluation of fragmentation follows. The numbers presented below are created summing up the results of each execution for each allocator creating an average behaviour.

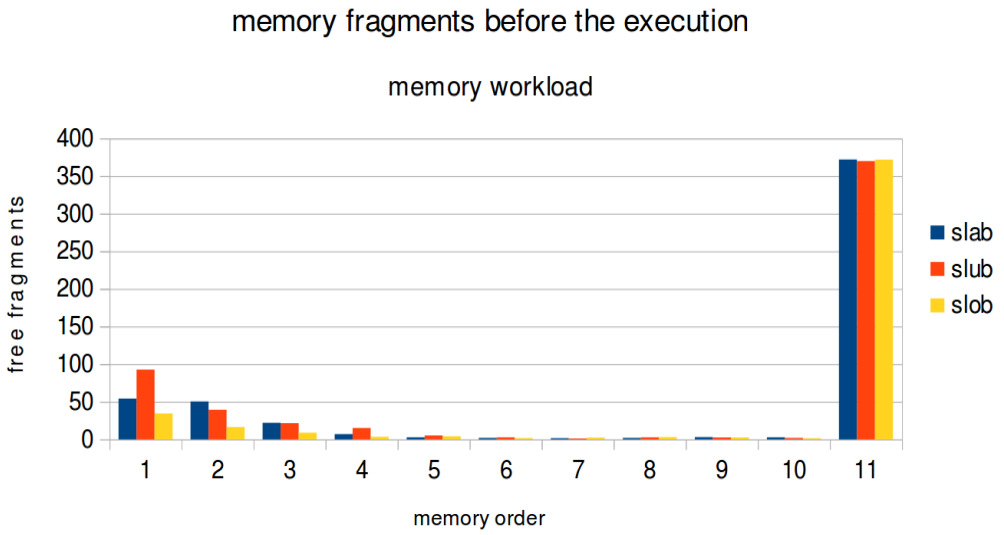

Figure 7a: arm64 free fragments of memory before the memory workload

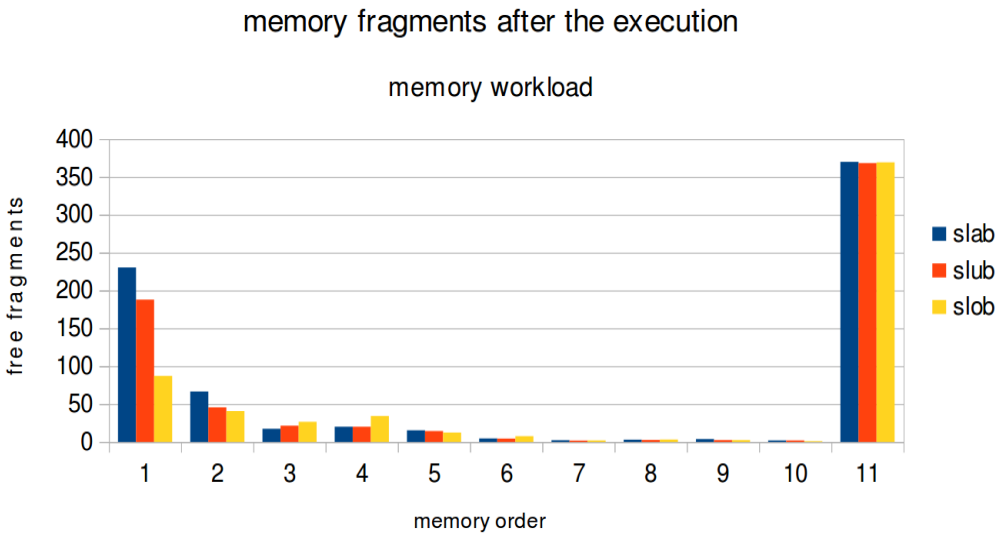

Figure 7b: arm64 free fragments of memory after the memory workload

Figure 7 represents the situation of the free memory fragments before and right after the

execution of the memory workload. Easily we can see that the free memory fragments after the execution of the memory benchmark with slab allocator are increased more than with the other allocators. Slub allocator as the figure shows causes big increment of the free fragments too. Slob is the allocator with the smallest fragments increment. It must be explained that the evaluation of the fragmentation is not fully absolute. More specifically during the execution of the workload if the fragmentation becomes big and big memory allocations start failing the `mm_page_alloc_extfrag` event occurs. During this event memory allocations are reallocated with a way that will reduce the fragmentation caused. In the above diagrams and also the diagrams that follow the fragmentation evaluation takes place comparing the fragments before and after the execution of the workload. However the existence and the call of the above fragmentation event is absorbed in the time performance of each allocator since the more this event is called the more the time grows and therefore the bandwidth performance of the allocator reduces.

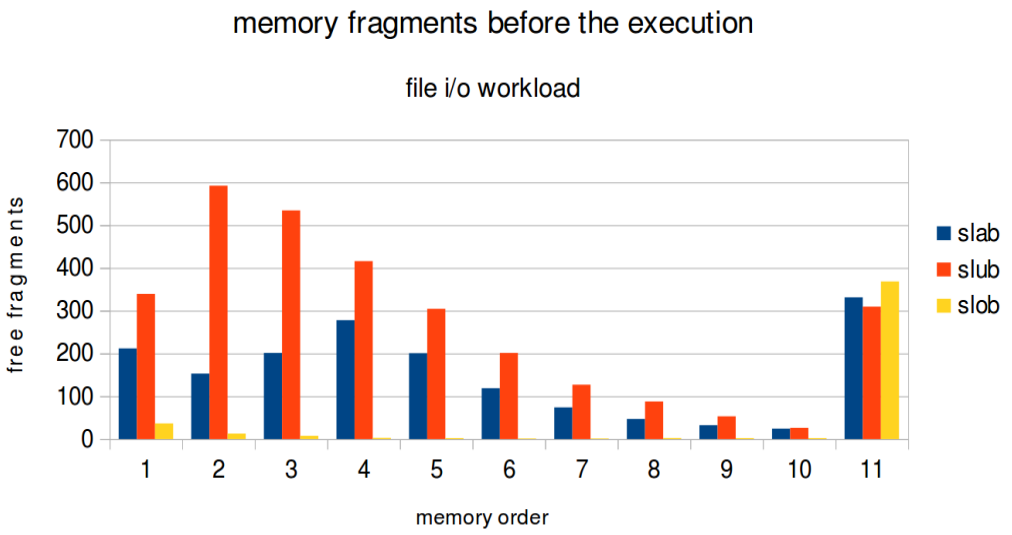

Figure 8a: arm64 free fragments of memory before the file i/o workload

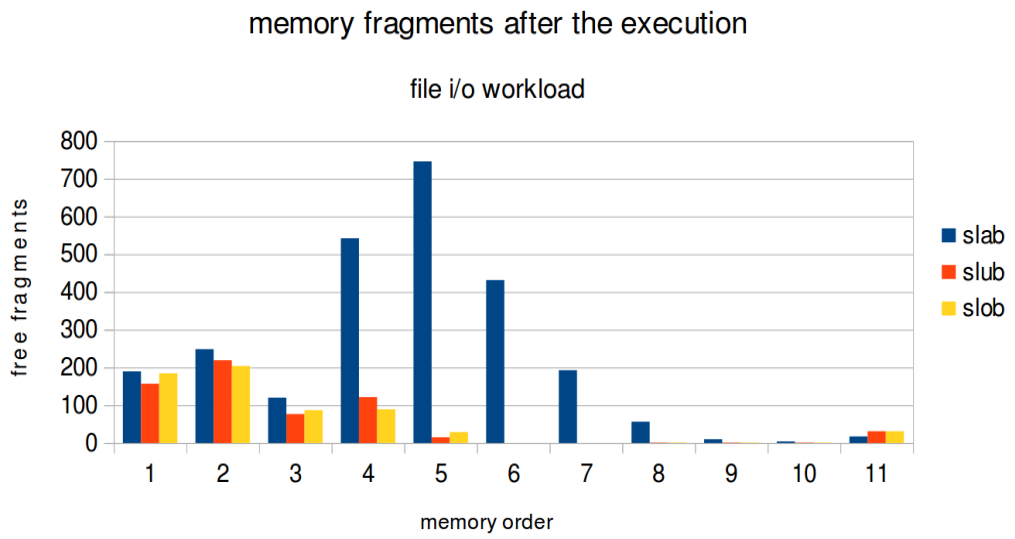

Figure 8b: arm64 free fragments of memory after the file i/o workload

Figure 8 containing free memory fragment diagrams follows. In the first diagram slub seems to start with a big number of fragments. The values in the above diagrams are the averages of all executions. In general slub's free fragments before the workload were a lot less than the above diagram shows. The increment of the average number of the available fragments was caused by the third execution when slub allocator after the restart of the system started with a huge amount of free fragments. Specifically while the average number of free fragments before the execution of the workload with the third execution not included was:

70.5	41.75	13.25	6	2	2.25	2.25	2.25	3.25	1.5	368.75

during the third execution slub allocator started with the below amount of fragments:

1415	2794	2619	2056	1514	997	627	430	253	125	73

increasing a lot the average values. After the execution of the file i/o benchmark the biggest amount of free fragments remained using the slab allocator. Slub's performance follows with the amount of free fragments after the execution using slob to be the smallest.

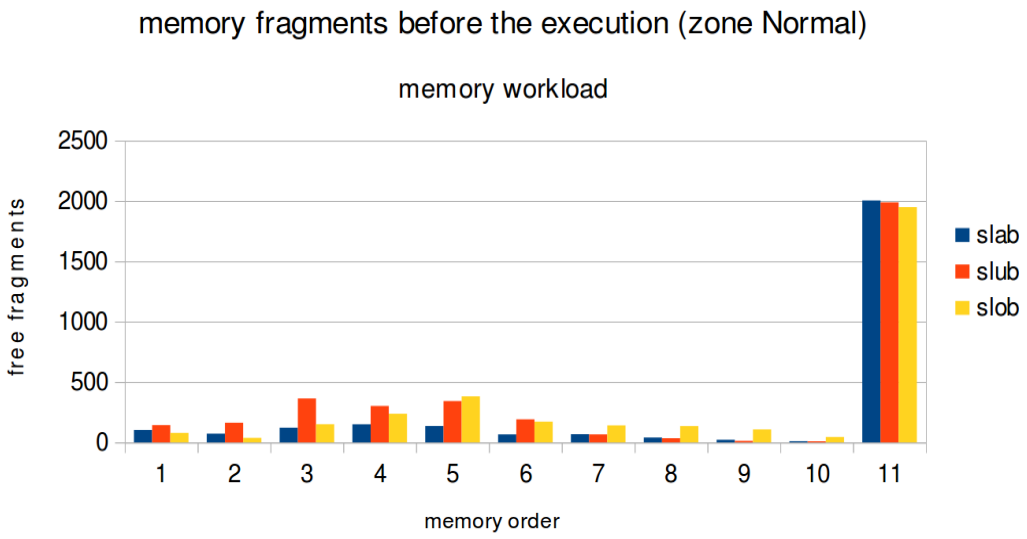

Figure 9a: x86 free fragments of memory before the memory workload

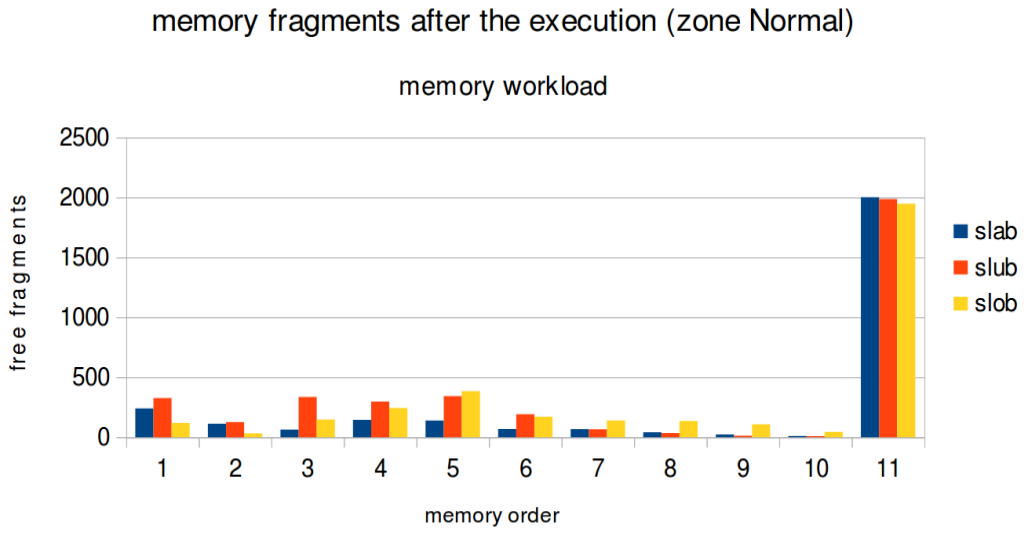

Figure 9b: x86 free fragments of memory after the memory workload

Lets continue with the free fragments in x86 system. Figure 9 shows the free fragments before and after the execution of the memory workload. The diagrams above show only the situation in zone Normal of the memory since zoneDMA and zoneDMA32 remain unaffected from the memory benchmark. Looking at the diagrams its not easy to see which allocator creates more free memory fragments during the execution. Slub seems to leave the most fragments after the execution while the fragments before and after using the slob allocator remain almost the same. The free fragments after the benchmark using the slab allocator are less than the slub's fragments and more than the slob's fragments.

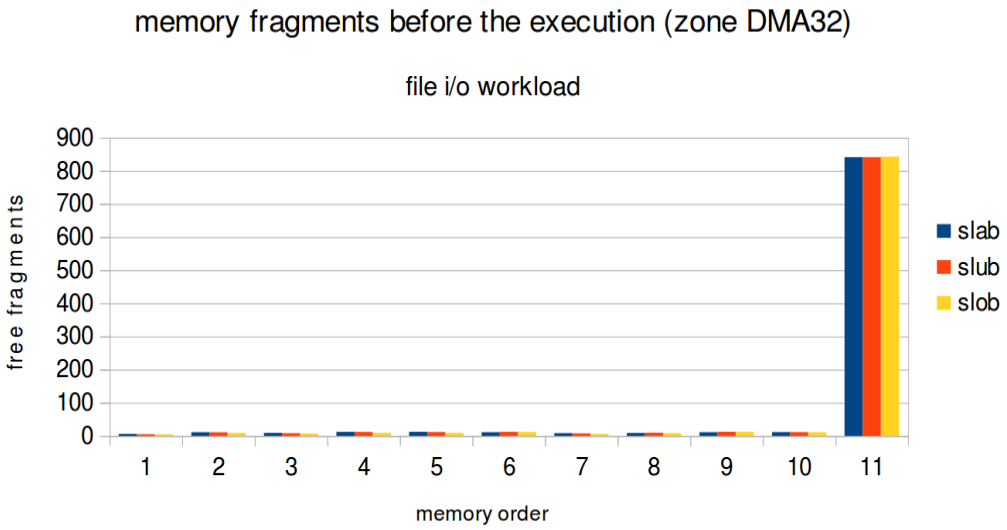

Figure 10a: x86 free fragments of memory before the file i/o workload

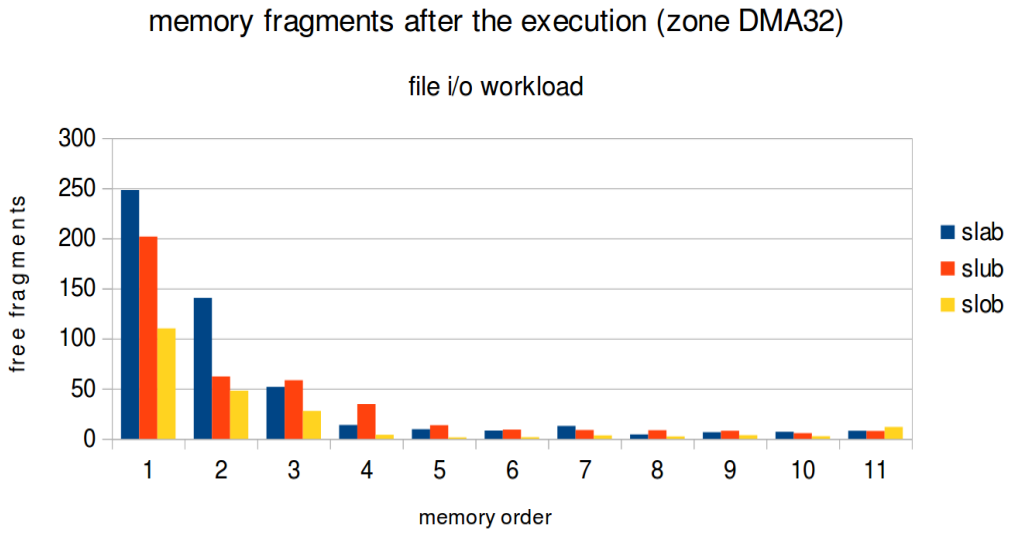

Figure 10b: x86 free fragments of memory after the file i/o workload

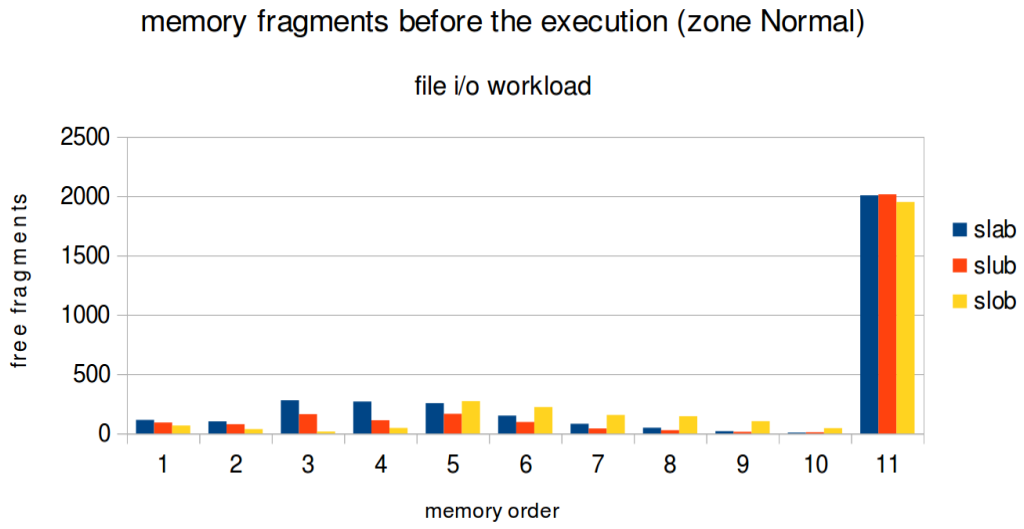

Figure 10c: x86 free fragments of memory before the file i/o workload

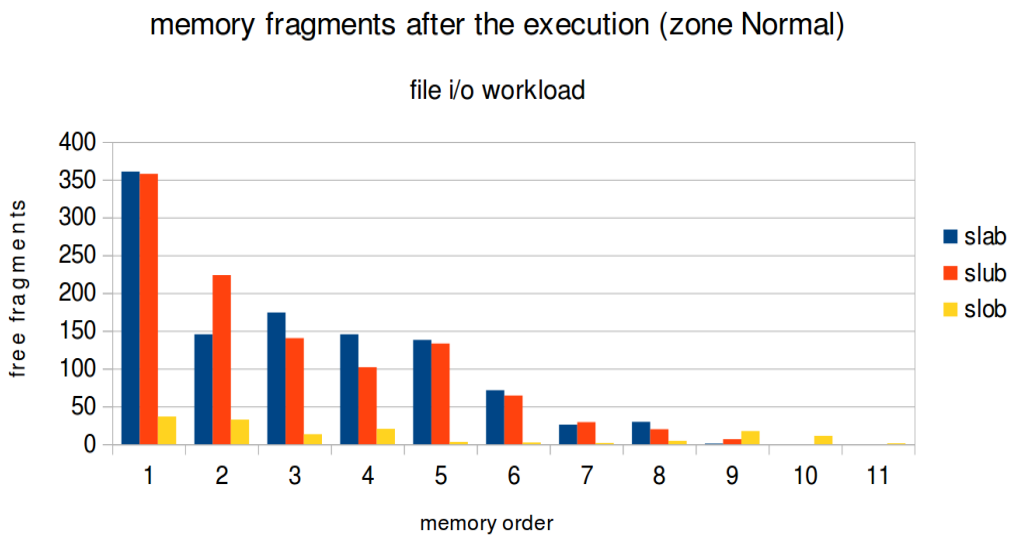

Figure 10d: x86 free fragments of memory after the file i/o workload

Finally figure 10 shows the free fragments before and after the file i/o workload execution on the x86 system. ZoneDMA of the memory is not included at the figure since no changes of the free fragments occurred during the execution. In zone DMA32 the amount of the remaining free memory fragments after the execution of the file i/o workload are the biggest using the slab allocator. The use of the slub allocator caused less increment of the memory fragments than the slab but still the offset between the slub and the slob number of fragments is big. The small amount of fragments remain during the execution using slob as explained above is probably caused by the call of the `mm_page_alloc_extfrag` event.

Conclusion

arm64	Best allocator		Worst allocator
Time (memory workload)	slob	slab	slub
Time (file i/o workload)	slab	slub	slob
Consumption (memory workload)	slub	slab	slob
Consumption (file i/o workload)	slab	slob	slub
Final fragments (memory workload)	slob	slub	slab
Final fragments (file i/o)	slob*	slub*	slab
x86	Best allocator		Worst allocator
Time (memory workload)	slob	slub*	slab*
Time (file i/o workload)	slab	slub	slob
Consumption (memory workload)	slab*	slub*	slob
Consumption (file i/o workload)	slub	slab	slob
Final fragments (memory workload)	slob*	slub*	slab*
Final fragments (file i/o)	slob	slub	slab

Figure 11: Table with sorted allocators

Figure 11 contains sorted allocators according to their average performance for each benchmark as presented in the above figures 1-10 (the average performances of the allocators marked with "*" are close in the particular benchmark). As the table shows slob in both systems achieved better time performance executing memory operations (memory workload). Furthermore the

memory fragments after the execution of memory and file i/o workloads seem to be less using slob comparing to the free fragments left by other allocators. As explained before the lower number of free fragments after the execution using slob allocator does not mean that this allocator causes necessarily less fragmentation since during the execution of the workload could caused more fragmentation recovery events ending up with less fragments. Also one more similarity in the behaviour of the allocators in the used systems is that slab achieved better time and therefore bandwidth performance than the other allocators executing i/o operations in arm64 and x86 system. Looking at the fourth column of the table representing the worst allocators' performances slob achieved the worst time performance in file i/o benchmark and consumed the most memory during the memory benchmark in both systems while slab left in average the biggest number of fragments after the execution in all workloads and execution systems. In general the behaviour of the allocators are similar in both systems considering that many offsets of the performances between the executions using one allocator on x86 and on arm64 systems are caused by different hardware architecture and components.

The main differences between the used systems are that in memory workload executed on arm64 system slub consumed less memory than slab while on x86 slab consumption was the least. Also after the file i/o benchmark on arm64 system slab was the more efficient allocator in terms of consumption but on the x86 system slub was slightly better. However the difference between the performance of the allocators is not always absolute. In almost all benchmark executions, except the memory workload execution on arm64 system, the best performance of the worst allocator is better than the worst performance of the best allocator as the error windows showed. That means that one other sequence of a benchmark executions could lead to a bit different results. This way the above table (Figure 11) represents the average behavior of the allocators in 5 random executions.

Considering the above similarities and differences together with the figure 11 it’s easy to say that there is no a perfect allocator but instead each allocator achieves better performance from the other allocators in some of the benchmarks. Specifically the slob allocator achieved the best time performance under in-memory pressure in both systems but at the same time was the allocator which consumed the biggest amount of memory during this workloads. On the other hand during file i/o benchmarks slab was the allocator with the best time performance on both systems while slub achieved slightly worse performance and slob was the slower allocator. At the same time on the arm64 system slab consumed the smallest amount of memory while slub caused the least memory consumption on the x86 system. It seems that there is a trade off between the time, the memory consumption and the fragmentation caused between the three allocators. This way during the selection of an allocator all the advantages and disadvantages of each allocator should be considered to achieve the best possible choice depending of the needs of their use.

References

1) Understanding the Linux Virtual Memory Manager by Mel Grooman

2) Introduce the SLOB allocator (https://lwn.net/Articles/157944/)

3) The SLUB allocator (https://lwn.net/Articles/229984/)

4) Sysbench manual (http://imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf)

5) Sysbench (https://wiki.gentoo.org/wiki/Sysbench)

6)Understanding the Proc File System (http://www.linuxfocus.org/English/January2004/article324.shtml)

7) proc buddyinfo (https://www.centos.org/docs/5/html/5.1/Deployment_Guide/s2-proc-buddyinfo.html)

To access the scripts used and detailed files with the execution stats visit: https://github.com/aleksandergar/thesis