Assignment 3 - Report

Philipp Küppers Sofie Vos

October 2022

1 Summary of the article by Kamp

In the article Kamp complains about the fact that even though the concept of virtual memory is around for over 40 years, most implementations do not make use of this concept. He noticed that most of implementations do not take into account the anisotropic memory access delay caused by virtual memory, CPU caches, write buffers, and other facts of modern hardware, which have a significant impact on their performance.

This can result in cases where even though the speed of algorithms and data structures might be optimal in theory, the real world speed can be drastically worse. He shows this in an example where a binary heap is supposed to be optimal, however it gets beaten by a factor of 10 in real world in a small number of cases by a B-heap which leverages the concept of virtual memory. Those edge cases turned out to be the norm, which made the implementation using a B-heap a lot faster.

2 Expanding first generation in a B-heap

We look at the situation where a child branches and the children are in a new page. As we can see for example in Figure 6.B, the first generation of the new page with nodes 10 and 11 does not expand. This is the case on the other B-heap pages as well.
One reason is that when the children in the parent page expand to a new page, we want to have both children in the same page. The main idea behind this is that since we traverse the tree vertically, we always compare children with its parent, which means that we would have more page operations if they are on different pages. As we have learned from the book and the lectures, more page operation produce more page faults which would slow down the processes a lot. Another reason is that if we would expand the first generation, we would require more pages since the pages could not be properly filled in order to adhere to the reason before and we would waste two spots per page.

Since each page size is a power of 2, it would work for two page sizes. Each page has at least two elements. If we then immediately branch, we would get power's of 2 new elements every level until expanding would require a new page. However when we look at how many elements we have in the end, we need to add two elements to fill the page. The fix is to not expand the first generation, which let's us fill the page.

3 Test environment

For our measurements, we did not observe any hard page faults, because the program fits completely in physical memory.

3.1 Baseline measurements

user time:	28.084024 s
soft page faults:	524407
hard page faults:	0
max memory:	2100020 KiB
voluntary context switches:	0
involuntary context switches:	137
dummy value (ignore):	549755813888
CPU cycles (measured with perf) :	88487439811

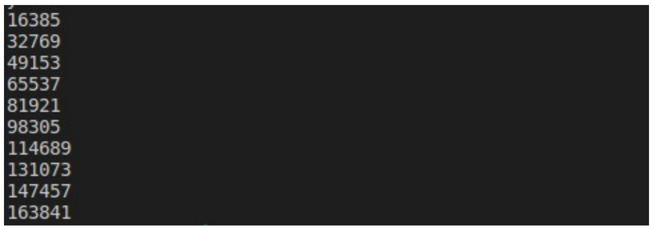
3.2 Test environment

Processor:	11th Gen Intel(R) Core(TM) i7-11370H @ 3.30GHz
Number of cores:	4
Number of threads:	8
Amount of RAM:	16GiB
Operating System:	Linux Ubuntu 22.04 LTS, dual boot
cache	L1 320KiB, L2 5MiB, L3 12 MiB
CPU time	clock: 100MHz

4 Changes

We observe that the program accesses the array with quite some gaps in between.

Here are the first few indexes accessed:



```
16385
32769
49153
65537
81921
98305
114689
131073
147457
163841
```

This means that for most of the iterations a new page gets accessed every time. This results in more possible page faults and significantly decreases the performance.

In order to fix this we looked at how the elements in the array are accessed and observed the following: When accessing elements at index `j * SIZE + i`, the outer for loop used `i`, while the inner for loop used `j`. Considering that the inner for loop increases to `SIZE` before the outer for loop gets incremented by one, we can see there are huge spaces left in memory which explains the output we showed earlier. This results in a lot of different pages having to be accessed every iteration because the indexes are so far apart at first. Fortunately we can fix this by swapping `i` and `j` in lines `42` and `43` and `58` and `59`. The changes can also be seen in the `main.cpp` files we handed in.

5 Final measurements

user time:	7.989869 s
soft page faults:	524403
hard page faults:	0
max memory:	2099984 KiB
voluntary context switches:	0
involuntary context switches:	43
dummy value (ignore):	549755813888
CPU cycles:	37421064804

The most remarkable differences compared to the baseline is that the user time is around 20 seconds shorter and that the number of CPU cycles are reduced by 51066375007. Also the involuntary context switches and soft page faults have reduced. The main reason is that with the improved implementation, we access fewer different pages. The array does not fit on one page, which means that we still have some soft page faults. Furthermore we wanted to point out that even though our test set up did not have any hard page faults, our changes would have reduced them significantly to speed up the program because of the fewer pages used.