0. Names and Student Numbers

Matthijs Muis, s1066918

1. Summary of the article by Kamp

Kamp explains that theoretically comparing the runtime complexity of the binary heap as to the B-heap, missed the crucial detail that the binary heap's locality of reference is much worse than that of the B-heap. This means that in practice, the binary heap causes much more thrashing than the B-heap, and the B-heap outperforms the binary heap when virtual memory is taken into account in simulations.

Binary heaps are layed out in consecutive virtual memory like an array, where the node at place `n` has children at places `2n+1` and `2n+2`. The binary tree is layed out in memory breadth-first. But for most algorithms running on the binary heap, such as `heapify`, `pop_front`, `push_heap`, `is_heap` etc, we traverse the heap depth-first, often comparing parents to children or siblings (for example, pushing down an element into the heap requires finding a max/min among a parent-child-child triple). At deeper levels of the heap, this means that entire pages are paged in and later out again just to access a single element in that layer (layers are subsets of a single generation). This is detrimental for cache reuse and for utilization in virtual memory.

The B-heap has a memory layout that fills the pages vertically to match the direction by which the heap is traversed: it contains more generations in a single page, and this satisfies the needs of most heap algorithms, which don't care about comparing elements across a generation, but only care about parents with children and siblings.

The arrangement in B-heaps increases the average number of comparisons and swaps needed to maintain the heap invariant. In particular because the tree fails to expand for one generation, so it has a more irregular shape so to speak. Therefore, theoretically, the B-heap is outperformed by the binary heap. Yet operations stay within one VM page and as a consequence reduce the average number of page faults/cache misses.

Because this difference outweighs the algorithmic overhead in practical applications with virtual memory or caching, where locality of reference is crucial to not have too many page faults and cache misses, B-heaps actually quite often outperform binary heaps.

2. Explanations

- **why the first generation in a B-heap does not expand:** It is assumed that each VM page can contain 8 items of the datatype stored in the heap. The heap operations that we want to optimize for, i.e. `heapify`, etc, require the comparison of child - parent and siblings (the 2 children with the same parent). We want to group parents with both children on one page, and since 1 parent and 2 children are already 3 items, we can put 2 such triples on a page (6 items), but since it would be nice if we can fill the page (internal fragmentation, see), we just add a generation between the parents and their children where we do not expand. The heap invariant for a B-heap is still the same relation between parent and child: that a parent should be larger than its child or children.
- **An explanation of whether this works for all page sizes:** No, it is just because Kamp assumed 8 items per page, or at least he assumed 8 items needed to be grouped together in a block, and pages

can be divided evenly into such blocks. If it were for example 10 items, it would be more useful to have 3 generations in a frame, expand the first generation of children and don't expand the second generation:

```
  8       9
10  11  12  13
14  15  16  17
```

The point is that we try to keep as many parents with their children, and also keep as many siblings (belonging to the same parent) together. By together I mean together on a page, so that the comparison of child and parent which happens so frequently, does not require a memory access off the page (or, off the cache (line)), which would reduce utilization through paging overhead. This we want to achieve, while also filling the page with as many elements as possible. This implies, for some item numbers, that some generations do not expand.

The assumption of 8 items on a page (or a power of two in any case) is not so strange, if you understand that most datatypes are aligned on a power of 2 bytes, and pages contain a power of 2 bytes, so the nr. of items on a page is probably some power of 2 as well. With 1, 2 and 4 items, you cannot even fit multiple generations properly on a page, so 8 seems to be the minimum number needed to make some use of the b-heap's memory layout. 16 and larger powers of 2 are just multiples of 8, so this layout can then just be repeated multiple times on one page.

To summarize: we want two things:

1. keep subsequent generations and siblings of the same parent together on a page.
2. have "a power of 2" items on the page to minimize internal fragmentation. So we group them together in blocks of 8 items, and this requires one generation that does not expand. If every generation expands, you will never get a power of 2: 2 + 4 + 8 + 16 + .. + 2^k will never get you a 2^n, because that will equal 2^{k+1} - 2 (which is never a power of 2 since it is divisible by the odd number 2^k - 1).

3. Baseline measurements, description of the machine

The provided code was run without any alterations, i.e.

- `SIZE = 16384`
- `REPEAT = 1`

This gave the following baseline metrics:

```
user time:                   21.158217 s
soft page faults:            524403
hard page faults:            0
max memory:                  2099840 KiB
voluntary context switches:  0
involuntary context switches: 106
dummy value (ignore):        549755813888
typical page size:           4096
```

Running the same code for 10 times gave an average running time of about 20.7 seconds, with a minimum of 18.695 seconds and a peak of 21.961 seconds. `soft page faults` varied between 524402 and 524403, `hard page faults` was always 0. There were never involuntary context switches, voluntary constext switces varied between 106 and 292. The remainder of metrics (page size, max memory) did not change. They seem to be constants defined by either the OS or the program.

Environment/Machine info

Hardware model: Lenovo IdeaPad 5 15ALC05

- Processor: AMD Ryzen 7 5700u, 16 cores
- Main memory: 16,0 GiB

Operating system: Ubuntu 22.04.3 LTS, 64-bit

`lshw` gives, among other things, the cache sizes and capacities:

L1 cache

- size: 512KiB
- capacity: 512KiB

L2 cache

- size: 4MiB
- capacity: 4MiB

L3 cache

- size: 8MiB
- capacity: 8MiB

Unsurprisingly, they were all already completely filled.

4. List of made changes and motivations

1. Obvious change: the order of indexing in the first loop (lines 26-31) The inner loop is over j, meaning that subsequent memory accesses are `SIZE * 8` bytes apart. These strides can be much smaller, improving locality of reference and thereby reducing cache misses & paging operations. if we just interchange the for-loop headers, the operation done for each (i,j) does not change in semantics because the order of the operations does not matter (we only write a value generated entirely by the CPU to a location in memory, not relying on data from memory).

Measurement to motivate:

```
user time:                   16.404895 s
soft page faults:            524404
hard page faults:            0
max memory:                  2099840 KiB
voluntary context switches:  0
```

```
involuntary context switches: 90
dummy value (ignore):         549755813888
typical page size:            4096
```

Page faults did not decrease, but user time did (user time is used as a proxy for caching efficiency).

2. Can we do the same in line 43/44? Yes, because in the operation in the loop, res[] only depends on img[], so whether we order the operations i-first or j-first does not change the semantics.

Motivating measurement:

```
user time:                    13.117643 s
soft page faults:             524404
hard page faults:             0
max memory:                   2099968 KiB
voluntary context switches:   0
involuntary context switches: 449
dummy value (ignore):         549755813888
typical page size:            4096
```

3. Can we do the same in line 55/56? Yes, because `dummy` will in the end just hold the sum of all res[SIZE * i + j] for each (i,j)-pair once. So whether we iterate over i first or over j first, does not matter.

Motivating measurement:

```
user time:                    2.261544 s
soft page faults:             524402
hard page faults:             0
max memory:                   2099968 KiB
voluntary context switches:   0
involuntary context switches: 8
dummy value (ignore):         549755813888
typical page size:            4096
```

Note that the dummy value is still 549755813888. This does not *prove* that interchanging indices does not change the semantics of our program, but in any case it does not disprove it!

4. In the averaging operation, notice that `j` and `l` are the "long-stride"-indices, indexing over stepsize `SIZE`, while `k` and `i` are the short-striding indices (step size 1). But `l`'s loop is nested inside `i`'s loop, and we can reorder this, because the operands are taken from a different variable in memory than where the result is written (`img` and `res` respectively). I hoped to improve the performance even more by interchanging `i` and `l`.

Note however, that just interchanging the loop headers for `i` and `l` will create some redundant operations and mess up the semantics:

- `res[j * SIZE + i] = 0;` becomes nested inside the loop for `l`, causing it to be executed 2 redundant times (for `SIZE * SIZE` (i,j) pairs, so quite a lot of redundant operations)
- `res[j * SIZE + i] /= 9;` is also executed for iterations of the `l`-loop, which whill change the semantics of the program: we now effectively divide every entry by 9^3 = 729.

To remove the redundancy and restore the semantics, I put these operations in separate nested loops before and after the convolution:

```
for (int64_t j = 1; j < SIZE - 1; j++) {
    for (int64_t i = 1; i < SIZE - 1; i++) {
        res[j * SIZE + i] = 0;
    }
}
```

```
for (int64_t j = 1; j < SIZE - 1; j++) {
    for (int64_t i = 1; i < SIZE - 1; i++) {
        res[j * SIZE + i] /= 9;
    }
}
```

It is not visible that the performance improved so much upon reordering.

An example run gave:

```
user time:                   2.152935 s
soft page faults:            524403
hard page faults:            0
max memory:                  2099968 KiB
voluntary context switches:  1
involuntary context switches: 10
dummy value (ignore):        549755813888
typical page size:           4096
```

The number of soft page faults did not decrease. But this number has also not decreased on previous improvements 1,2,3. Actually, soft page faults are not the page faults we care about (see below): thrashing is about hard page faults. But there are no hard page faults anyway, so we have to look at the caching behaviour of the program. I will do this below using the `perf` tool.

Again, the semantics seem to be unharmed, judging from the dummy value, but this is not a proof.

I had expected this modification to give another improvement in reducing page faults and cache misses, because the operation

```
res[j * SIZE + i] += img[(j + l) * SIZE + i + k];
```

can now be done for constant j and l while i and k iterate, instead of iterating k and l, where l takes a big step and thus requires more paging/caching.

However, it apparently doesn't quite work out that well. A possible cause is that l only iterates over -1, 0, 1, so 3 values, which means that per i, only 3 extra big strides are taken in the implementation where the `l`-loop is nested inside the `i`-loop. Since 3 is not so much as `SIZE`, which was accounted for in 1,2,3, the speedup is not so great.

5. Final measurements and comparison with baseline measurements

We notice several things w.r.t. paging:

- The number of soft page faults is enormous and does not decrease with the alterations. This is not a surprise: soft page faults are caused by pages that are in memory but the process simply doesn't have a valid mapping to yet. This happens all the time, for example with shared libraries such as `libc`. So it is not a wonder that there are many of these soft page faults. Moreover, we see that such page faults do not decrease when we ameliorate the thrashing of our program, because thrashing is related to hard page faults: thrashing occurs when a process is replacing frames that are needed again right away, because the process does not have enough frames and needs to do subsequent memory accesses in a lot of different pages. This implies hard page faults (pages are paged out of main memory and into again), not soft ones.
- The number of hard page faults in the original program was already 0 or 1, so much improvement could not be made anyway. There was barely any thrashing behaviour on the paging level. The program needs two arrays of `SIZE * SIZE * 4` bytes (float is 4 bytes) = 1 GiB each, given my RAM of 16.0 GiB this is not a problem, even with a browser with 12 tabs open running in the background.
- But there certainly *was* thrashing on the caching level: these are an analogon of main memory for main memory, but are much smaller, ranging from 512 KiB to 8 MiB. Re-ordering the loops decreased the user time 10-fold. The `rusage` struct does not contain any direct information about cache misses. For that, let us use `perf`, in particular I used the command

```
sudo perf stat -B -e cache-references,cache-
misses,cycles,instructions,branches,faults,L1-dcache-load-misses,L1-dcache-
loads,L1-dcache-stores ./memory
```

for both the original program and the final alterated program included in the submission. I did not use it on each subset of changes, but just on the alteration with the lowest running time (i.e. changes 1,2,3,4).

The original program:

```
 Performance counter stats for './memory':

     9.132.228.071      cache-references
(71,41%)
     1.442.900.421      cache-misses                     #   15,800 % of
all cache refs         (71,42%)
    95.026.237.991      cycles
(71,43%)
```

```
    29.894.120.136      instructions                     #    0,31  insn
per cycle              (71,44%)
     5.244.504.944      branches
(71,44%)
           524.386      faults
     6.245.810.161      L1-dcache-load-misses            #   31,15% of all
L1-dcache accesses   (71,43%)
    20.052.100.483      L1-dcache-loads
(71,42%)
   <not supported>      L1-dcache-stores

      21,993361182 seconds time elapsed

      21,019269000 seconds user
       0,971966000 seconds sys
```

Final program (changes 1.-4.):

```
user time:                   2.120256 s
soft page faults:            524394
hard page faults:            0
max memory:                  2099968 KiB
voluntary context switches:  1
involuntary context switches: 10
dummy value (ignore):        549755813888
typical page size:           4096

 Performance counter stats for './memory':

       399.528.753      cache-references
(71,33%)
        14.552.943      cache-misses                     #    3,643 % of
all cache refs         (71,46%)
    14.012.869.839      cycles
(71,50%)
    25.385.853.661      instructions                     #    1,81  insn
per cycle              (71,50%)
     5.237.294.479      branches
(71,50%)
           524.384      faults
       204.065.808      L1-dcache-load-misses            #    3,02% of all
L1-dcache accesses   (71,42%)
     6.760.202.905      L1-dcache-loads
(71,30%)
   <not supported>      L1-dcache-stores

       3,256922595 seconds time elapsed

       2,120478000 seconds user
       1,136256000 seconds sys
```

Finally, I am interested to see whether the alteration under 4., i.e. swapping the loops of `i` and `l`, has improved the locality of reference of the program. For the other alterations, it is obvious.

Program with changes 1.-3.

```
user time:                   2.033967 s
soft page faults:            524395
hard page faults:            0
max memory:                  2099968 KiB
voluntary context switches:  1
involuntary context switches: 8
dummy value (ignore):        274877906944
typical page size:           4096

 Performance counter stats for './memory':

       299.054.140      cache-references
(71,37%)
        14.561.311      cache-misses                    #    4,869 % of
all cache refs         (71,49%)
    13.257.823.868      cycles
(71,49%)
    24.988.001.538      instructions                    #    1,88  insn
per cycle              (71,49%)
     5.234.261.302      branches
(71,49%)
           524.386      faults
       151.518.959      L1-dcache-load-misses           #    3,13% of all
L1-dcache accesses   (71,40%)
     4.840.784.365      L1-dcache-loads
(71,28%)
   <not supported>      L1-dcache-stores

       3,087525167 seconds time elapsed

       2,034206000 seconds user
       1,053142000 seconds sys
```

It turns out my argument under 4. was wrong! Trying it again 10 times consistenly yields about 151.000.000 cache misses for the program with alterations 1.-3. and about 200.000.000 cache misses for the program with alterations 1.-4.

I think that my argument under 4. overlooked that when we restore the semantics of the program by adding two extra loops, i.e.:

```
  for (int64_t r = 0; r < REPEAT; ++r) {

    // Extra loop:
    for (int64_t j = 1; j < SIZE - 1; j++) {
      for (int64_t i = 1; i < SIZE - 1; i++) {
```

```
          res[j * SIZE + i] = 0;
        }
      }

      // Original convolution loop:
      for (int64_t j = 1; j < SIZE - 1; j++) {
        for (long l = -1; l < 2; l++) {
          for (int64_t i = 1; i < SIZE - 1; i++) {
            for (long k = -1; k < 2; k++) {
              res[j * SIZE + i] += img[(j + l) * SIZE + i + k];
            }
          }
        }
      }
      for (int64_t j = 1; j < SIZE - 1; j++) {
        for (int64_t i = 1; i < SIZE - 1; i++) {
          res[j * SIZE + i] /= 9;
        }
      }

      // Extra loop:
      for (int64_t j = 1; j < SIZE - 1; j++) {
        for (int64_t i = 1; i < SIZE - 1; i++) {
          dummy += res[j * SIZE + i];
        }
      }
    }
```

We forget that *adding two extra loops* will introduce new cache misses, because the program again has to iterate over the entire array, of which the beginning is not in cache anymore at the start of the next (i,j)-loop.

So the file I will submit is `main.cpp` with alterations 1.-3.. For your better understanding of alteration 4., I have also added `main4.cpp` which contains alterations 1.-4. Feel free to look at it, if you already understand what I tried to do from my explanation under 4., then you can leave it there.

5. and explain any remaining page faults from the theory in the book.

I think I already discussed this en passant. We have in the occasional run one hard (=major) page fault but usually zero. Major page faults happen when page is referenced but it is not in main memory (or it may be in memory, but compressed. What I mean by "not in main memory" is "not in the physical form of an active page, inside main memory"). These faults could be due to page-outs forced by competing page-ins of other processes, or it could be due to the kernel prepaging one frame too few initially, meaning that that missing frame will lead to a hard page fault when it is referenced later on.

There are still many soft page fault (=minor page faults) in the process of running the final program (524395, a lot), and this is completely unrelated to the locality of the program (and thus also unrelated to its caching behaviour and performance). Soft page faults occur when a frame is already present in memory but the process simply has no logical mapping yet. The theory in the book gives two causes of this:

1. a process references a shared library that is in memory but does not have a mapping to it in its page table. This is very likely to happen at least once in the run of the process, since we use at least stl and libc functionality from at least 7 included headers, of which libc frames are likely to be present in memory already because so many programs (that are running next to our program) make use of it. The book indicates that the low number of major faults and high number of minor faults is also an indication that our program makes use of shared libraries, otherwise all its imports would have lead to more hard page faults than 1 on average.

2. The second cause occurs when a page is reclaimed from a process, but not yet zeroed out and assigned to another process, and the original process references it again. I.e. when you write something on a whiteboard, leave it there, expect it to be erased after the break, but it happens to still be there so you reassign the whiteboard to yourself and don't have to put everything back on it again (the example is a bit superficial). I don't think this type of minor page fault occurs a lot in our running program, since it is the kind of minor fault that is actually nearly a hard fault, so if this were to happen a lot, then occassionally we would also see more hard faults when the process is not so lucky to be able to reassign itself a reclaimed page. But we barely have any hard page faults...

To summarize: I believe that most of the page faults that still occur can be explained as being a soft page fault due to creating logical references to shared libraries. Both the original program and the improved program need only about 2 GiB of physical memory, and this is at least in accordance with the low number of hard faults, and this is also why I expect that most of the soft faults are due to type 1. and not type 2. causes.