Operating Systems and Architecture (CO527) Spring 2016

Caching

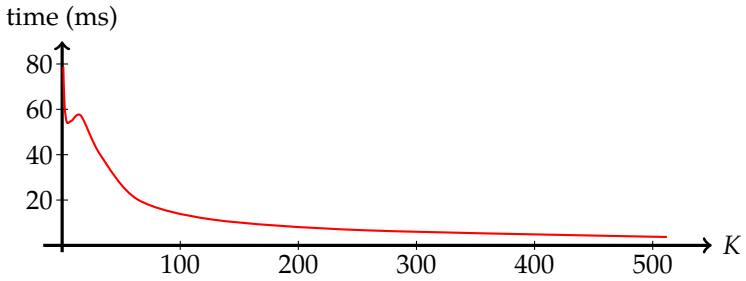
Fred Barnes and Radu Grigore

Cache Effects

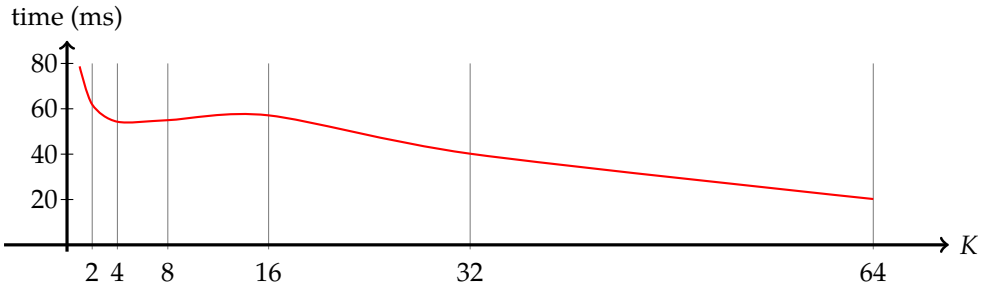
In the previous lecture, we saw how malloc and free are implemented on top of arenas, which are big chunks of memory obtained via mmap. We saw that the amount of work done by malloc is not constant: it depends on the state of the arena. To figure this out, we assumed that the arena behaves like an array. In particular, we assumed that accessing an element of the array takes constant time. Things are not so simple, as is revealed by the following experiment. Consider the following code:

```
1  for (int i = 0; i < (1 << 27); i += K) arena[i] += 3;
```

Here, arena is an array of 2^{27} integers obtained via mmap. The number of operations is proportional to $2^{27}/K$, so we expect the runtime to be $\sim 1/K$. It is:



Except that something funny happens for very small K. Let's take a closer look there.

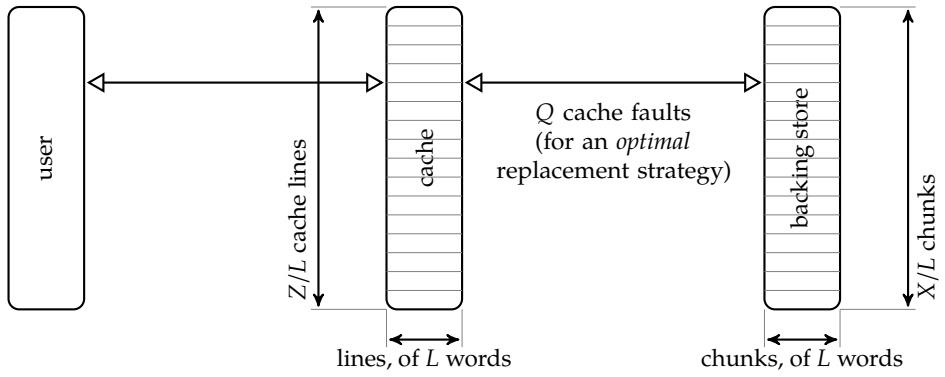


The runtime goes down by 20% when K goes up from 1 to 2, but then it appears to stay constant as K further increases. Only after K passes the threshold of 16 does the runtime begin to decrease again.

Here is a plausible explanation. Recall, from the architecture lectures, that memory is organised in a hierarchy: at one end we have fast and small memory (registers), at the other end we have slow and big memory (hard drives). The information between two adjacent memory levels is transferred in chunks. The key to understanding the behaviour observed above, is to realise that what matters is *how many* chunks are transferred between different levels of the hierarchy. In other words, the runtime of the program from above is dominated by cache faults. To understand cache faults, let us look in more detail at caches.

Ideal Cache Model

We follow the definition of an ideal cache model from [3]. The (Z, L) *ideal cache model* is illustrated in the following figure:



The user thinks it interacts with a backing store which is an array of X words. So, the user issues a sequence of commands: read (tell me what word is at index i) and write (put word w at index i). A cache handles these commands, pretending it is the backing store. The cache can often answer faster than the backing store would because it has a fast local storage: a set of Z/L arrays of size L, which are called *cache lines*. Correspondingly, the backing store is split into chunks of length L: chunk k is a sub-array of the backing store from index kL to index $(k+1)L-1$. Each cache line is in one of $1 + X/L$ possible states: either it is empty, or it is mapped to some chunk k. (The meaning of 'line ℓ is mapped to chunk k' will become clear later. For now, just think that line ℓ has a tag that says either 'mapped to k' or 'empty'.) The cache can change the state of a cache line via two operations:

1. If a cache line ℓ is empty, then copying the content of chunk k into line ℓ makes line ℓ mapped to chunk k. We say that chunk k was *loaded* into line ℓ.
2. If cache line ℓ is mapped to chunk k, then copying the content of line ℓ into chunk k makes the line empty. We say that line ℓ was *evicted*.

The cache ensures that distinct lines are not mapped to the same chunk.

Although not part of the definition of an ideal cache, let us make some remarks about the typical values of L, Z, and X in practice: $L > 1$ and $X \gg Z$. (1) Cache lines hold more than word in order to exploit *locality of reference*, which is the tendency of memory accesses that are close in time to also be close in space. (2) If a cache could hold all the data in the backing store, then the cache would not be needed. Even more, because caches tend to be so expensive compared to the backing store, they are significantly smaller.

Suppose now that the user tries to access chunk k. The cache will:

1. Ensure that there is a line ℓ which is mapped to chunk k. It does so as follows:
   (a) if there exists a line ℓ that is mapped to chunk k, then there is nothing to do [cache hit]
   (b) otherwise [cache miss (or *fault*)]
      i. if all lines are nonempty, then pick a line ℓ and evict it
      ii. load chunk k into some empty line ℓ
2. Access line ℓ, as if it was chunk k.

It remains to specify how to pick which line to evict, in Step 1(b)i; that is, it remains to specify a so called *replacement algorithm*. The usual definition of the ideal cache model [3] says that cache lines are evicted in such a way that the total number of cache faults is minimised. The intention of this definition is to free us from thinking about any particular implementation of the cache, when we analyse a program that uses the cache. This is good because often actual implementations of the cache are complicated and because often we don't even know what the implementation is. But, what if the minimum number of cache faults is significantly

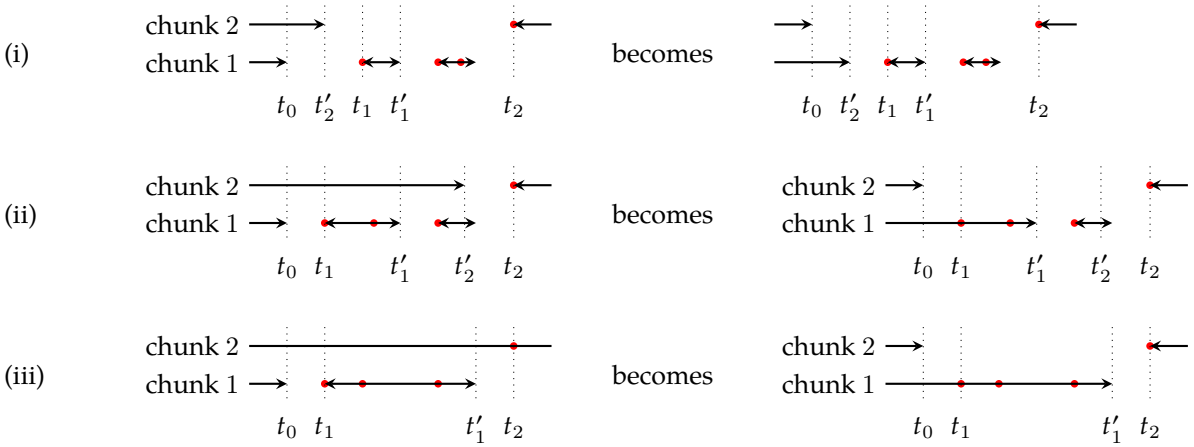


Figure 1: Illustrations for the proof of Theorem 1. An arrow ⟷ represents the time there exists a cache line mapped to a certain chunk. Faults occur when a chunk is loaded in memory, so they are represented by the left ends of the arrows ←. A red circle • represents an access to a chunk.

smaller than the number of faults occurring with the implementation on our computer? This is unlikely, because even simple replacement algorithms are, roughly speaking, within a constant factor of the theoretical optimum. To make this statement precise, we need to look at a few specific replacement algorithms.

- **MIN**: evict the line which is last to be referenced again (in the future)
- **FIFO**: evict the oldest line, where age is measured since the line was loaded (**f**irst **i**n **f**irst **o**ut)
- **LRU**: evict the line that was **l**east **r**ecently **u**sed.

MIN replacement is just as good as the ideal cache.

Theorem 1 (Belady 1966)**.** *The replacement algorithm MIN achieves a minimum number of cache faults.*

Proof.* Consider a run of the cache that does not correspond to MIN replacement. If we follow the eviction decisions taken in the given run, then we find a first point in time where the run evicts a different line than MIN replacement would. Let n be the number of cache faults in the given run. We will show how to produce another run that agrees with MIN replacement on a strictly longer prefix. Moreover, the number of cache faults in the modified run will be $\leq n$. If we apply our transformation repeatedly, then eventually we obtain a run that agrees with MIN replacement. Moreover, the number of cache faults in this run will also be $\leq n$. We can start this process with any run, including with one that has a minimum number of faults. It follows that at least some run that agrees with MIN replacement is optimal. It's easy to see that all runs that agree with MIN replacement have the same number of cache faults, and hence all runs that agree with MIN replacement are optimal.

It remains to describe the transformation mentioned above. Let t_0 be the smallest time where the given run disagrees with MIN replacement. Without loss of generality, let $t_0 = 0$ and let the chunk being accessed be chunk 0; moreover, suppose the run evicts (a line mapped to) chunk 1, and MIN replacement is consistent with evicting chunk 2. Let t_1 be the time when chunk 1 is accessed again, and let t_2 be the time when chunk 2 is accessed again. Let t'_1 be the time when the chunk loaded at time t_1 is evicted. Let t'_2 be the time when chunk 2 is first evicted. So,

$$
\begin{aligned}
t_1 &= \min\{\, t \mid t > 0 \text{ and chunk 1 is accessed at time } t \,\} \\
t_2 &= \min\{\, t \mid t > 0 \text{ and chunk 2 is accessed at time } t \,\} \\
t'_1 &= \min\{\, t \mid t > t_1 \text{ and chunk 1 is evicted at time } t \,\} \\
t'_2 &= \min\{\, t \mid t > 0 \text{ and chunk 2 is evicted at time } t \,\}
\end{aligned}
$$

We take $\min \emptyset = \infty$. Because MIN replacement is consistent with evicting chunk 2 at t_0, but not consistent with evicting chunk 1 at t_0, we know that $t_1 < t_2$.

Instead of evicting chunk 1 at t_0, we evict chunk 2 at t_0. We need to say when chunk 1, which is now in memory immediately after t_0, is evicted: if $t_2' < t_1$, then we evict it at t_2'; if $t_2' \geq t_1$, then we evict it at t_1'. We need to say when chunk 2, which is now not in memory immediately after t_0, is loaded: at t_2.

We do not overflow the cache: chunk 1 is in a line for some extra time, but only before $\min\{t_1, t_2'\}$ when chunk 2 occupied a line in the given run. We do not have more faults: The only fault that could be introduced by our transformation is at t_2, and that happens in the case $t_2 < t_2'$ in which case a fault disappears at t_1. Finally, the transformed run evicts a line only if the original run evicted a line at that time. □

TODO: simplify proof

Figure 1 shows several instances of the transformation described in the proof from above.

MIN replacement may be just as good as the ideal cache, but it is also just as un-implementable. The following result tells us that two implementable replacement algorithms, FIFO and LRU, are not much worse than MIN.

Theorem 2 (Sleator and Tarjan 1985)**.** *Consider a fixed but arbitrary sequence of memory accesses. Let f be the number of cache faults for the MIN replacement algorithm with a (Z, L) cache. Then the number of cache faults for the FIFO (or LRU) replacement algorithm with a $(2Z, L)$ cache is $\leq 2f$.*

The proof [5] is not as easy as that of Theorem 1.

Informally, Theorem 2 says that the FIFO and the LRU replacement algorithms are almost optimal. So, let us look at some examples of how FIFO and LRU work.

FIFO and LRU

This is an example of how a FIFO cache functions:

$$[] \xrightarrow{2} [2] \xrightarrow{0} [20] \xrightarrow{1} [201] \xrightarrow{2} [201] \xrightarrow{3} [013] \xrightarrow{1} [013] \xrightarrow{0} [013] \xrightarrow{0} [013] \xrightarrow{3} [013] \xrightarrow{3} [013]$$

It looks complicated, but all you need to do is understand the notation. A bracketed expression like [201] means that the cache contains three nonempty lines that are mapped, respectively, to chunks 2, 0, and 1. We also assume that each nonempty cache line has a timestamp, and that cache lines are listed in increasing order of their timestamps. For example, the cache line that is mapped to chunk 2 has the oldest timestamp. A transition like $\xrightarrow{2}$ represents an access to chunk 2. Finally, the chunk is red, as in $\xrightarrow{2}$, when the access caused a cache fault.

We call the string of chunks that are accessed the *reference string*. In the example above, the reference string is 2012310033. Let us see how does LRU handle the same reference string, using again a cache with three lines:

$$[] \xrightarrow{2} [2] \xrightarrow{0} [20] \xrightarrow{1} [201] \xrightarrow{2} [012] \xrightarrow{3} [123] \xrightarrow{1} [231] \xrightarrow{0} [310] \xrightarrow{0} [310] \xrightarrow{3} [103] \xrightarrow{3} [103]$$

Note that the chunk last accessed appears at the right side of the cache: look for the pattern $\xrightarrow{x} [\ldots x]$.

For both FIFO and LRU, the line that is evicted comes from the left side of the cache, in our notation. Exploiting such regularities, it is possible to give a common description to FIFO and LRU. Both FIFO and LRU associate a timestamp with each cache line. When a line needs to be evicted, both FIFO and LRU choose the cache line with the oldest (smallest) timestamp. FIFO and LRU differ only in how they set the timestamps. FIFO sets the timestamp of a cache line when that line is loaded; LRU sets the timestamp of a cache line every time that line is accessed.

The previous description of FIFO and LRU emphasizes the similarity of the two replacement algorithms. However, this similarity is at a conceptual level — we do not want to actually implement FIFO and LRU using timestamps, because it would be too slow.

Virtual Memory

In the ideal cache model, the backing store is an array. The backing store is never used directly by the user, though. So, as long as the cache does its pretending well, there is no need to have an actual, real array as the backing store: a virtual one suffices. On a 64 bit system, each process views the memory as one array of 2^{64} bytes: this is the so-called *process address space*. (We will see what processes are later. For now, it's safe to think of 'process' as a fancy word for 'program'.) Nobody has that much RAM; and even if someone had that much RAM, no process would use it all. To convince yourself, follow this simple calculation. Let's say we have a 3 GHz CPU that can write one word to memory in each clock cycle. How many bytes could possibly be written to memory in one year? The answer is $3 \times 10^9 \times \frac{64}{8} \times 60 \times 60 \times 24 \times 365$, which is less than 2^{60}. Thus, a process needs to run more than 16 years before it can use all its address space, and that's if all it does is to write to previously unwritten memory and if all the writing is extremely fast. So, the process address space is sparsely populated. And, of course, only the populated parts need to be stored in the physical memory.

Computers divide virtual and physical address spaces into chunks called *pages*, usually of 4 KiB. A *page table* tracks the correspondence between pages in the virtual memory and pages in the physical memory. Only those pages from the virtual memory that are in use need to have a corresponding page in the physical memory. Processes are allowed to use (that is, read and write) a page only after they announced they intend to use it. They do so, by calling mmap, directly, via malloc, or via some other function. For example, the call

```
1 mmap(0, length, prot, flags, -1, 0);
```

prepares several pages of the process address space for later use: these pages are now a so-called *mapped area*. The number of pages is big enough so they contain at least length bytes. The returned value is the smallest address of the mapped area. Different mapped areas have different characteristics, chosen by setting prot and flags to appropriate values. For example, an area could be shared (with other processes) or it could be private.

When an area is mapped, the kernel makes an internal note that the new area exists, and not much else happens. If a process tries to use a (virtual) page and that page has no correspondent in the physical memory, then the operating system finds to which area does the page belongs and calls the *page fault handler* associated with that area. The page fault handler allocates a page in the physical memory, and records, in the page table, the correspondence between the virtual page and the physical page. If there is no space in the physical memory (or there is very little space), then a *swap* process is invoked which moves pages from the physical memory to the hard drive. So, swapping out pages is quite similar to evicting cache lines.

Because it involves the hard drive, swapping is slow. Under normal workloads, this is not a problem because swapping happens rarely. Each process typically has a small portion of its virtual memory that is hot, meaning that it was accessed recently. Swapping occurs typically when the hot region changes: what cooled down gets moved to the hard drive. However, it does happen occasionally that there is more hot virtual memory than there is physical memory. This could happen if there are many processes running in parallel, or if a process actively tries to use a huge portion of its virtual memory. When such a situation occurs, swapping is invoked frequently, and its slowness becomes important: The computer becomes unresponsive and the hard drive is used continuously. This condition is called *thrashing*.

Let us now look at the common case, in which a process accesses a virtual page that has a corresponding physical page. A process operates with addresses that refer to its own virtual memory: so-called *virtual addresses*. But, somehow, the computer must access the corresponding physical page, which has a *physical address*. How is the virtual address translated to a physical address? Briefly, one needs to do a lookup in a page table. For example, consider a 64 bit machine with 4 KiB page size, and the virtual address $\texttt{fedcba9876543210}_{16}$. The position within the page is given by the last 12 bits: $\texttt{210}_{16}$. The virtual page number is given by the other 52 bits: $\texttt{fedcba9876543}_{16}$. Suppose now that there are 4 GiB of physical RAM, and the page table contains the mapping $\texttt{fedcba9876543} \mapsto \texttt{edcba}$. Then the corresponding physical address is $\texttt{edcba210}$. In general,

$$p = page\_table\left(\left\lfloor \frac{v}{s} \right\rfloor\right) \cdot s + v \bmod s$$

where p is the physical address, v is the virtual address, and s is the (fixed) page size. Often, $s = 2^k$ for some k, which means one can use this C code:

```
p = (page_table[v >> k] << k) + (v & ((1 << k) - 1));
```

This translation, from a virtual address into a physical address, needs to be done every time a process accesses memory. This translation is therefore performance critical, which is why most systems have hardware support for it: MMU (**m**emory **m**anagement **u**nit).

Conceptually, the page table is a function from virtual page numbers to physical page numbers. A simple representation for such a function is a set of mappings; for example,

$$\{\texttt{fedcba9876543} \mapsto \texttt{edcba},\ \texttt{fedcba789abcd} \mapsto \texttt{12345},\ \texttt{1111111111111} \mapsto \texttt{22222},\ \ldots\}$$

Given the left-hand side of a mapping, one wants to quickly find the right-hand side. Even with hardware support, this operation is expensive. Note that one cannot simply have an array indexed by the (virtual) page number: that would be too big. In the example above, we'd need an array of size 2^{52}. Instead, one uses a multi-level page-table:

$$\left\{\texttt{fedcb} \mapsto \{\texttt{a9876} \mapsto \{\texttt{543} \mapsto \texttt{edcba}\},\ \texttt{a789a} \mapsto \{\texttt{bcd} \mapsto \texttt{12345}\}\},\ \texttt{11111} \mapsto \{\texttt{11111} \mapsto \{\texttt{111} \mapsto \texttt{22222}\}\},\ \ldots\right\}$$

In this example, we first lookup the most significant 20 bits of the page number, and the result is a nested page table; in the nested page table we lookup the next 20 bits of the page number, and the result is again a nested page table; finally, in the doubly-nested page table we lookup the last 12 bits of the page number, and the result is the physical page number (aka, the *page frame number*). Thus, we can implement the top-most level of the page table as an array of size 2^{20}, which is a much more reasonable size for an array. The drawback is that we need *three* lookups in such arrays.

To speed up the address translation even more, the hardware takes advantage of locality of reference. Not only memory accesses exhibit locality of reference, but also lookups in the page table. Here is one way to understand this concept. Suppose that the (nested) page table contains N mappings. Suppose that doing N lookups takes 1 second. Now we do an experiment: we sample at random 1 second from the operation of the page table. Because of our assumptions, in this time the MMU did roughly N lookups. Now we look at how many *distinct* mappings where looked up. Let's say the number is M. Locality of reference means that often it is the case that $M \ll N$. Clearly, it's worth caching the page table!

Indeed, the MMU usually contains a piece of hardware that caches page table entries and is called TLB (**t**ranslation **l**ookaside **b**uffer). The TLB is implemented as a content addressable memory. This means that the TLB tracks a set of mappings, can do fast (constant time) lookups, and yet it does not use an array. Instead, the hardware is designed in a completely different way. Wait a minute — you might say — if such a thing is possible to implement in hardware, why don't we use it to store the whole set of mappings? The answer is that content addressable memories are expensive. Simply put, we know how to implement TLBs, but their size is limited by what we can achieve in hardware.

Exercises

1. ► Explain the behaviour of the example from the section *Memory Hierarchy*. [Hint: Think of cache lines.]
2. Ostrovsky [2] does *not* observe a decrease in runtime when K goes from 1 to 2. Does the runtime decrease on your computer or not? [Hint: use the program p3.c that accompanies these notes.] Can you find an explanation for the different behaviours?
3. The ideal cache model evicts lines by copying their content to the chunk they map. When can the cache avoid copying, without affecting correctness? How would you implement this optimisation?
4. Why is MIN replacement un-implementable?
5. In which situation, MIN replacement does *not* uniquely determine the line to evict? Does the choice of line affect the number of cache faults? [You probably need to do this exercise if you want to understand the proof of Theorem 1.]
6. Does LRU ever cause more cache faults than FIFO?

7. ► Consider a cache with 4 lines and the reference string 16200953943797562744. Simulate the FIFO replacement algorithm by hand. Simulate the LRU replacement algorithm by hand.

8. Implement the FIFO replacement algorithm efficiently.

9. Implement the LRU replacement algorithm efficiently.

10. Prove Theorem 2.

11. The Linux kernel has a vm_area_struct for each mapped area within a virtual address space. If mmap is asked to map a new area, and no hint is given as to where the new area should be placed, then how does Linux chose the location? [Hint: See the function unmapped_area in the file mm/mmap.c.]

12. In the Linux kernel, page allocation is done using the buddy system. Look at the implementation in mm/page_alloc.c. An older version of the implementation is described here.

13. Look at [8] to see how content-addressable memory is implemented.

References

[1] Luu, What's New in CPUs Since the 80s and How Does It Affect Programmers?

[2] Ostrovsky, Gallery of Processor Cache Effects

[3] Frigo et al., *Cache-Oblivious Algorithms*, 1999

[4] Belady, *A Study of Replacement Algorithms for Virtual Storage Computers*, 1966

[5] Sleator and Tarjan, *Amortized Efficiency of List Update and Paging Rules*, 1985

[6] Duarte, How the Kernel Manages Your Memory

[7] The mm/ directory of the Linux kernel version 4.2.0.

[8] Pagiamtzis and Sheikholeslami, *Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey*, 2005