Lecture 9 - Paging

Gidon Rosalki

2025-05-25

Notice: If you find any mistakes, please open an issue at `https://github.com/robomarvin1501/notes_operating_systems`

1 Reminder - memory management

The process feels it has access to the entire memory, and there is address translation that maps its virtual memory to the physical memory. We have the Memory Management Unit, which takes the virtual addresses from the CPU, and translates them into physical addresses. We implement this by dividing the process address space into fixed size pages (usually 4KB), and divide the physical memory into frames of the same size. This way, any page can be mapped directly to any frame, which prevents external fragmentation, and the mapping of pages to frames can be stored in a page table. This way, we need not map the entire address space, only the parts of it that we are using.
The page table maps pages to frames. We have a separate page table for each process, and we switch tables as part of a context switch. The page table is populated, and managed, by the OS, and reflects the decisions of what should be mapped where. It is used by the MMU to perform memory access at hardware speed.
I am neglecting a full example here of how the address translation works. There should be one in the previous lecture, and if not, it is made very clear in the slide for this weeks lecture.
The page tables take up a lot of space, and need to be stored in memory. We can improve this by optimising the page size, and using sophisticated page table structures (to be discussed later). Also, accessing the page table requires an additional memory access for every memory access. We resolve this overhead by caching recently used translations in the TLB (Table Lookup Buffer). This is an additional cache in the MMU, which recalls recently used mapppings between page numbers, and frame numbers. It is usually fully associative, with only about 64 entries, constructed from incredibly fast SRAM.
Since each process thinks it is running along, and may access the entire logical memory of the computer, the virtual memory is much larger than the physical memory. Fortunately, programs do not use all their address space all the time. For example, a program need not its initialisation code after it has finished initialising. It also may never need its error handling code. The unused parts do not need to be mapped to memory, and can thus be stored on the disk until they are needed. This reduces memory pressure, and allows more efficient process creation.

2 Virtual memory

The idea is virtualisation. We want to disconnect the limitations of the budget of our physical hardware, and make it look as though we have all the memory that we could want. The implementation of this is called **demand paging**. We bring pages to memory when we need them, and store them on the disk when we do not.
In essence, we have our pages in virtual memory. When a request is made from the virtual memory, a request is made to the page table. If there is a mapping to the physical memory, then it is pulled from there, and if there is not, we bring the page from the disk, and also store it in the physical memory. So now we extend our page table to include an entry of if the requested frame is currently available in the physical memory, or if it needs to be brought in from the disk. Remember, the physical memory can be considered as acting as a cache for the disk.
So for demand paging, the OS loads a page into memory only when it is needed. This way, we need less IO, and less memory, and have a faster response, and possibility of more users. Another option is pre-paging. This way, the OS guesses in advance which pages the process will need, and pre loads them into memory. This saves time if the OS guesses correctly, but comes at the cost of there being more overhead if it guesses wrong.

2.1 RAM as storage cache

The small, fast memory, serves as a cache for large, slow memory. We find things in the cache through address translation with the page table, and the TLB, and the demand pages should be stored in the cache. However, we need to figure out, which items should we evict from the cache, when it fills? We will discuss this extensively.

When the accessed page is not in memory, the CPU issues a virtual address that is in a page which is **not present** (or invalid). Therefore, the data cannot be accessed, so the MMU creates a PAGE FAULT exception, and the OS exception

handler is invoked. This initiates a disk operation to get get required data to a free frame. The process is put to sleep until it arrives, and other processors are run in this time. When the data arrives, it is assigned a frame number, and a valid bit, the process is awakened, and it reissues **the same instruction**, which will not generate an exception this time.

2.2 Page eviction

What if there is no free frame to which to map a given page? Then we need our page replacement algorithms, also known as page eviction policies. We need to select some mapped page, and evict it. To implement this, we need to copy its data to disk, and use the frame for the new page. Let us call this page the "victim".
So, how do we choose our victims? We could just choose at random, but then again, we don't especially want to risk evicting pages that are often used, since then we will just need to bring them back again shortly. So, we have many other posisble policies, such as:

- Optimal
- Random
- FIFO
- NRU (Not Recently Used)
- LRU (Least Recently Used)
- LFU (Least Frequently Used)
- etc.

Let us note something about the general implementation. Most algorithms refer to a frame list, but we need to know the page to evict. We have two options available:

1. Frame to page mapping (for example, an inverted page table, discussed in the next lecture)
2. Use the page table instead, where we ignore invalid pages, and keep a separate list of free frames, for each process.

For the rest of the lecture, we're going to ignore this issue, and just pretend that there is a frame list.

2.2.1 Optimal algorithm

Note that this algorithm is not feasible. It is also known as Bélády's algorithm, clairvoyant algorithm, MIN cache replacement policy.
The idea is to replace the page that **will not be used** for the longest period of time (and therefore the optimality of this algorithm can be proven).
Let us consider that we have space for 4 pages, and that our pages are used in the following order: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. So we start by filling up the pages with 1, 2, 3, 4, and then when we need to load 5, we load it in place of 4, since 4 will not be used for the longest time. When we need to reload 4, then we can unload 1 for example (since we have no information on the continuation). In short, we have 2 evictions. This is not feasible, since we need to know the future. This algorithm is only used for comparison, given another algorithm, how close it is to the optimal algorithm?

2.2.2 Random replacement

Evict any page randomly. This is the other extreme from optimal, optimal uses full knowledge of everything, including the future, and random uses no knowledge of anything, including the past. It is also only really used for comparison, given an algorithm, is it better than random?

2.2.3 First in First Out (FIFO)

The OS can sort the pages in the order that they were loaded into memory, for example by maintaining the order as a linked list. The victim is the first page (in this order), since it is the oldest page. This does however have the disadvantage that the page in the memory the longest, might actually have been used most frequently. As we can see, this does not store any information about usage, merely age. This was historically used in Windows NT, since it is independent of any hardware support, there is no need for special hardware to use it.

2.3 Bélády's anomaly (for FIFO)

The expected behaviour is that the larger the memory, then the fewer page faults. Consider the access series

$$1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$$

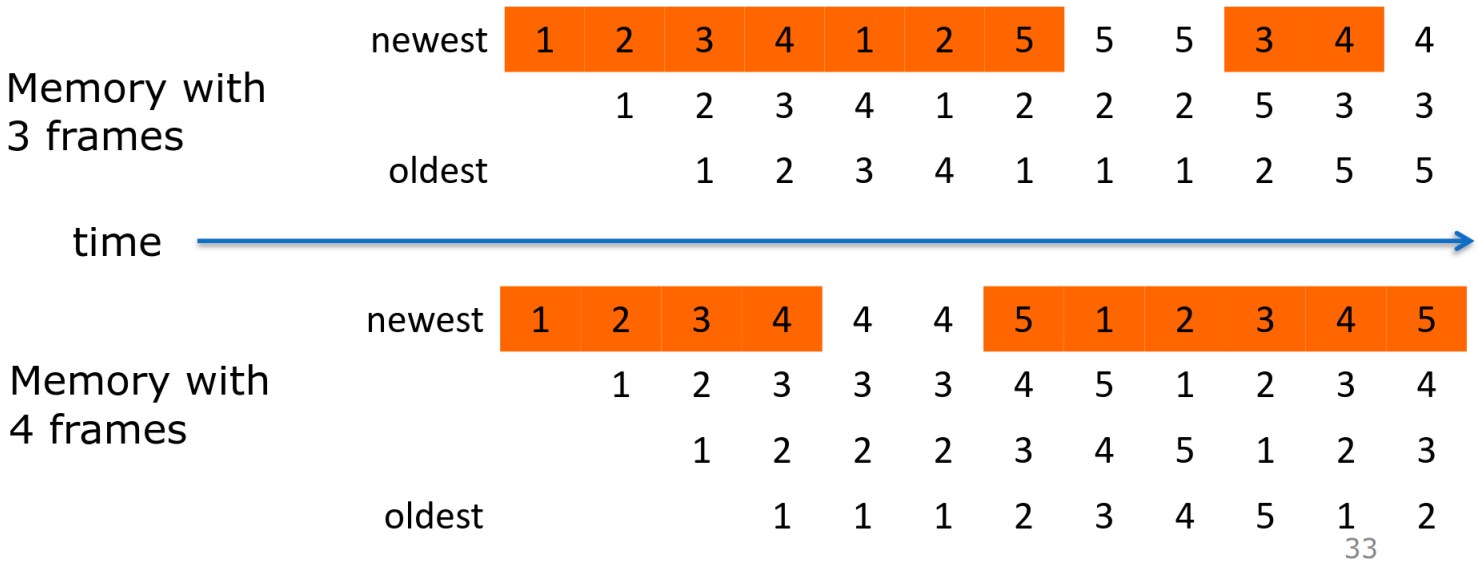

Figure 1: Demonstration of the anomaly

So we see that with 3 frames, there are 9 page faults, however, with 4 frames, there are 10 page faults. So we can see, that FIFO is not the best solution.

Let us consider, what do we really want? Ideally, we want to know the future, because we want to identify the process's **WORKING SET**. This is the set of pages that the process is currently using. Pages in the working set should be retained, and pages not in the working set can be evicted. This is based on the principle of **locality**.

So, how can we identify the working set? Let us begin by defining it formally.

Parametric definition: The set of pages, accessed in the last k memory accesses. With random access, the working set size $\approx k$. With locality, the working set size $<< k$. In practice, k is large, and we approximate via time, by asking which pages have been used recently.

We need hardware support for this. Memory access is done at clock speed, so we need hardware support to track it, but we also need to limit the overhead. We have a **reference bit**, also known as the used / accessed bit, which is turned on by the MMU when a page is accessed, and the **dirty bit**, also known as the modified bit, which is turned on by the MMU when a page is modified. In modern processors, these bits are stored in the page table, together with the frame number. In old processors, this was stored per frame.

2.3.1 Not Recently Used (NRU)

The concept is that we cannot know the future, so we will try and estimate it according to the past. We will use the pages **reference bits**, and periodically (on a clock interrupt), all reference bits will be cleared. When we need to choose a victim, we will choose randomly among the pages with a 0 reference bit. The logic is that these have not been accessed since the last clearing.

This solution is a crude (and expensive) estimation of LRU. We ignore a lot of data with it, and resetting all the reference bits is an expensive operation. Consider if we have 16GB of memory, then we have 4×10^6 bits to clear every time we want to reset the reference bits.

2.3.2 Least Recently Used (LRU)

Here we will try and use temporal locality, pages that were recently used are likely to be used again soon. We will evict the page that was least recently used. This gives a good approximation of the working set, and has the same logic as NRU, but has better accuracy. Unfortunately, implementing this is difficult, and expensive. If we save the information with timestamps, we need to store many bits per page, and then find the minimal timestamp on each eviction. If this is a sorted list, then we need to resort on every access. The list overhead is thus $\log_2(n)$ bits per page.

2.3.3 Clock algorithm

Also known as the second chance algorithm, here we think of all the frames as a circular list, with a hand pointing at one of them. We have a reference bit (in old architectures it was in each frame) in each page. When you need to evict a page, as long as the frame to which we are pointing has been referenced, we clear the bit and move on (give it a second change). We then evict the first non referenced page that we find.

The clock algorithm is still rather crude, it has no real discrimination between pages with different activity patterns. A better approximation of the working set is to track the time since the last reference, not only that it existed. We can do this by maintaining a crude timestamp per page, and thus have a working set approximation for the last x seconds.

2.3.4 WSClock algorithm (simplified)

BE WARNED: Tanenbaum's book (Editions 1 - 4) got this a bit wrong, and it was fixed in edition 5.

Here we maintain all frames / pages in a circular list, and maintain a virtual time variable (per process). Every time a page is referenced, we set the reference bit to 1, and set the time to the time it was references. Upon a page fault, if an eviction is needed:

1. We look at the frame / page to which the hand points
2. If the reference is 1, we set it to 0, update the time of last use, advance the hand, and go back to step 1
3. If the reference is 0 and the current virtual time - time of last use $< k$, then we advance the hand, and go to step 1
4. Otherwise, we evict the page

A dirty page is a page that was modified since the last time it was written to disk. The copy in the disk is different that the copy on the main memory (write back). When evicting the dirty page, we need to wait to write the page to the disk, and then wait for the new page to load into memory. When evicting a clean page, we just need to wait for the new page to load into memory. Therefore, we prefer to replace clean pages. Using this knowledge, we can improve our WSClock algorithm:

2.3.5 WSClock algorithm (full)

1. We look at the frame / page to which the hand points
2. If the reference is 1, we set it to 0, update the time of last use, advance the hand, and go back to step 1
3. If the reference is 0 and the current virtual time - time of last use $< k$, then we advance the hand, and go to step 1
4. If it is dirty (hardware), then advance the hand, and go to line 1
5. Otherwise, evict the page

If the page is dirty, but older than k, then schedule it for eviction, and the page is written to disk (cleaned) in parallel to the process actions (using DMA). If no candidate is found, then evict the oldest page, be it clean or dirty.

2.4 Global vs Local paging

When process P1 causes a page fault, should the OS choose a victim from one of P1 pages or from any process? Some operating systems have local paging (only from P1 pages) and some global ("best" candidate to evict)

2.5 Performance

Let us define p to be the probability of a page fault (disk access). Then

$$\text{Effective access time} = p \cdot (\text{page fault time}) + (\text{memory access time})$$

Since after the page fault we still need to load the data from RAM. Additionally

$$\text{Slowdown} = \frac{\text{Effective access time}}{\text{Memory access time}}$$

Generally, the page fault time is around $40\mu s$ on an SSD, with the memory access time of around 100ns. So for $p = 0.01$, we get a slowdown of 5, and for $p = 0.0001$, we get a slowdown of 1.04. Performance depends on locality. We need to ensure that costly disk operations are rare and amortize them across many memory accesses.