Lecture 10 - Paging and Locality

Gidon Rosalki

2025-06-08

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

1 Recap

1.1 Virtual memory

The idea is to disconnect from the limitation of our physical budget, and make it appear as though we have all the memory that we want. This can be implemented using demand paging, where we bring pages to memory when we need them, and store them on the disk when we do not. We store their addresses in the page table, and when we want to access memory, we request from the virtual memory to the page table, and if it's in the physical memory, it will be read from there, and if not, it will first be pulled from the disk. If necessary, it will remove a page in memory to make space for this new page. There are many methods to choose this page to remove, discussed previously.
The performance is found as follows: Let us call p the probability of a page fault, and thus the effective access time is given as:

$$\text{effective access time} = p \cdot \text{page fault time} + \text{memory access time}$$

Remember, we add disk load time to the memory access time. The slowdown is as follows:

$$\text{slowdown} = \frac{\text{Effective access time}}{\text{memory access time}}$$

These have typical numebrs (currently, in 2025) of a page fault time being rough;y $40\mu s$ (on an SSD), and access time of $100ns$. For $p = 0.01$, we generally get a slowdown of 5, and when $p = 0.0001$, we get a slowdown of around 1.04. Performance depends on locality, we should ensure that costly disk operations are rare, and amortise them across many memory accesses.

2 Workloads and Locality management

2.1 Reminder, principle of locality

Temporal locality: If we accessed a certain address, the chances are high to access it again shortly.
Spatial locality: If we accessed a certain address, the chances are high to access its neighbours.

Temporal locality actually reflects two separate phenomena. Firstly, when we access memory in a clustered manner, such as in the following order: AAAAABBBBBCCCCCDDDDD.
It also reflects skewed popularity: BABBBBCBBBBBABBBBBBDBCB

2.2 Measuring locality

We are faced with two main questions to measure locality: How do we know that locality really exists? How can we characterise the degree of locality given a certaina ddress stream? Measure the stack distance:

1. Scan the address stream
   (a) Search for each address on the stack. If it is found, then note its depth, and extract it
   (b) Push the address to the top of the stack
2. Output the distribution of depths at which addresses were found

This is the distribution of distances between successive accesses to the same address. Consider the following example:

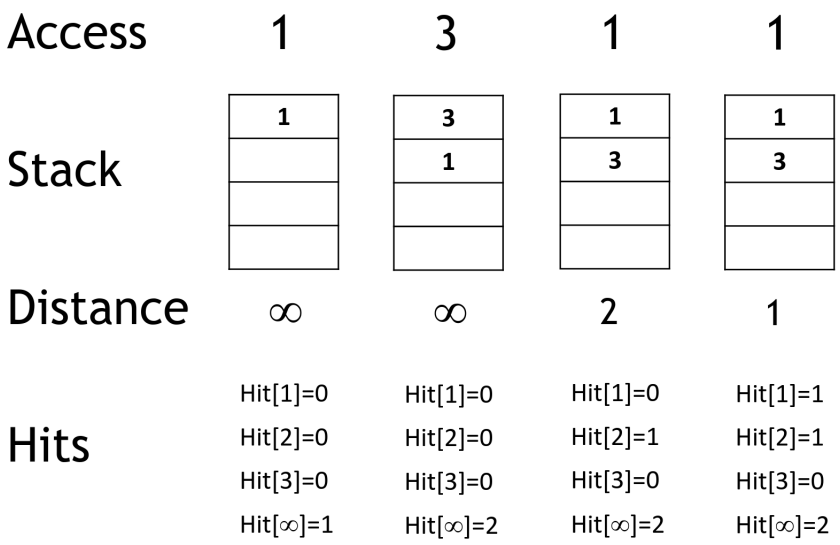

Figure 1: Locality measurement example

The stack distance performance is all we need to evaluate LRU performance. Let us assume a memory of size k. The top k items in the stack are the k most recently used items, and so they are the ones retained by LRU. So

$$\begin{aligned} p &= \text{Probability of a page fault} \\ &= \text{Probability of being in location } > k \text{ in the stack} \end{aligned}$$

3 Thrashing

Let us consider a definition:

- **CPU Usage**: The fraction of time the CPU is executing instructions

So we see that it follows naturally that multiprogramming increases CPU Usage. However, it also follows that as the degree of multiprogramming increases, the CPU usage increases, until a certain point, and then it falls off sharply to 0. This area is called **thrashing**: The system is so busy swapping pages in and out, that no process makes any progress.

Why does paging work? Due to the locality (of reference) model, that processes migrate from one locality to another, and that single process localities may overlap. So why does thrashing occur? When the size of the locality is greater than the total memory size (i.e., the combined working sets of all processes exceed the total memory capacity). The solution is to limit the degree of multiprogramming, we suspend processes when we exceed the degree of multiprogramming which we have defined as the limit.

We will resolve this by extending our process scheduler discussed before as follows:

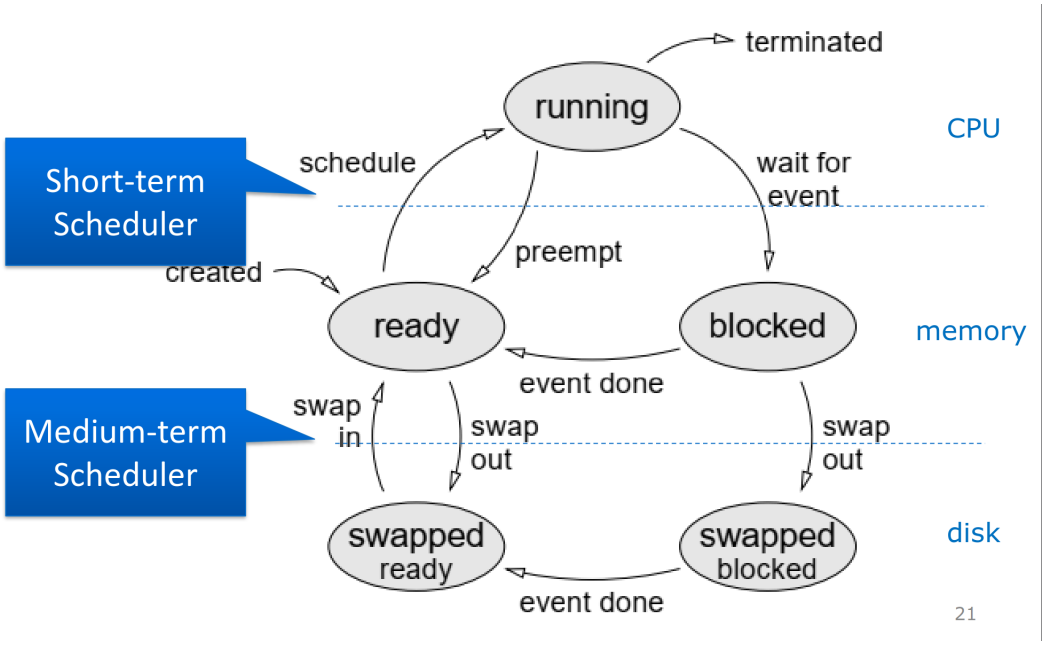

Figure 2: New scheduling

4 Hierarchical page tables

In 32 bit CPUs, if page size is 4KB, then each process has 2^{20} pages. If each page entry is 4B, then the page table itself requires 1000 pages, since page table size reflects the size of **logical** memory. So what happens in a 64 bit CPU?
Here we have 2^{64} bytes of logical address space (though many CPUs limit this to 2^{48} bytes). The page size is still 2^{12} bytes, and an entry in the page table is 2^3 bytes. For a 2^{48} address space, the page table itself is 2^{27} page. If we suppose that physical memory is 8GB $= 2^{33}$ bytes, then we need 2^{21} frames. This table does not fit into the RAM. At least $2^{27} - 2^{21}$ pages of the page table are completely invalid.

We can resolve this using Hierarchical Page Tables, where we break up the logical address space into multiple page tables. A simple technique is a two level page table. We can also use Hashed Page Tables, and Inverted Page Tables.

4.1 Hierarchical page table - structure

Let us consider a hierarchy 2 levels deep. Here, our logical address contains the page in the top level, the page in the second level, and the offset. The top level page points to a page in the top level PT, which in turn points to a frame in the second layer, where we have selected a PT by the page in the second level, contained within the logical address. From here, we construct the physical address as normal.
This way, instead of holding 1 table, with 1 million entries, we hold 1025 tables, with 1000 entries.

Let us consider a numerical example, with a 32 bit architecture. Let us consider accessing page 5200, with offset d. Well, in a hierarchy, let us suppose that page 5200 is held in frame 1000. The TLPT is held in frame 17, and the second level table 5 is held in frame 700. We go to frame 17, and read the 5th entry, whose value is 700. We then go to frame 700, and read the 80th entry, whose value is 10000. We then read the address 1000, with offset d.

In a 32 bit (x86) architecture, there tends to be 2 levels, the paging directory, and paging table, where we have two indices of 10 bits, and an offset of 12 ($2 \cdot 10 + 12 = 32$).
In a 64 bit architecture, (x64), There can be 4 levels, for 2^{48} physical bytes, 4 indices of 9 bits, and an offset of 12 ($4 \cdot 9 + 12 = 48$). There can also be 5 levels, for 2^{57} physical bytes, 5 indices of 9 bits, and an offset of 12.

So what is this good for? Well, considering 2 levels, not all the 2nd level page tables must be occupied. Therefore, they are created on demand, and the total page table size reflects actual page use, and not the theoretical address space.

4.1.1 Page creation

We begin with an all zero entry, where it is completely unmapped. There are 2 ways that pages are created:

- Explicit creation e.g. a heap request: A process invokes `mmap()` (such as by calling `malloc`), and the kernel assigns new page(s) for it, and adds entries to the page table(s)

- Implicit creation e.g. stack growth: The thread stack is not allocated its maximum size (8MB). When the stack grows to exceed its current maximum size, a page fault is called, and the kernel identifies the stack region, and assigns a new page for it, adding entries to the page table(s).

We will now note, that the MMU has more work to do, and harder work, needing to go between multiple different levels: Each virtual memory access translates into 3 (in 32 bit architecture) physical memory accesses. We also now have 2 page fault reasons:

- A non terminal page directory entry (e.g. entry in the TLPT) is not present / invalid
- Content page is not present (as before)

Most of the second level page tables are all invalid, so there is no need to store them **anywhere** (no, not even on the disk). When a page fault to such a table occurs, then the OS may decide to honour it, and create an all-zero frame for that page table, and store the entry needed + prepare a frame.
Fro a page table entry with a valid bit of 1, then the page is in memory. If the valid bit is 0 + some internal flags, then the page is swapped (on disk), and can be loaded with data from the swap table. If the page table entry is all 0s, then the page does not exist (no mapping).

4.1.2 NULL pointer

IGNORE THIS SECTION FOR THE EXAM, IT IS NOT RELEVANT TO THOSE QUESTIONS
What happens in runtime when we run `*ptr = 123;`, if `ptr = NULL`? How is this exception implemented? We don't want the compiler injecting code that checks every single pointer for being NULL, for every time we use a pointer, this would have a ridiculously high overhead. There is a very easy solution to this. We do not map page 0, and we "waste" 4KB (best 4KB ever wasted), and map NULL pointers to that page. Therefore, we get a hardware exception should we try and access a NULL pointer.

5 Other page tables

5.1 Hashed page tables

Here we refer to pages as a hash, having passed them through a hashing function. It follows the following diagram:

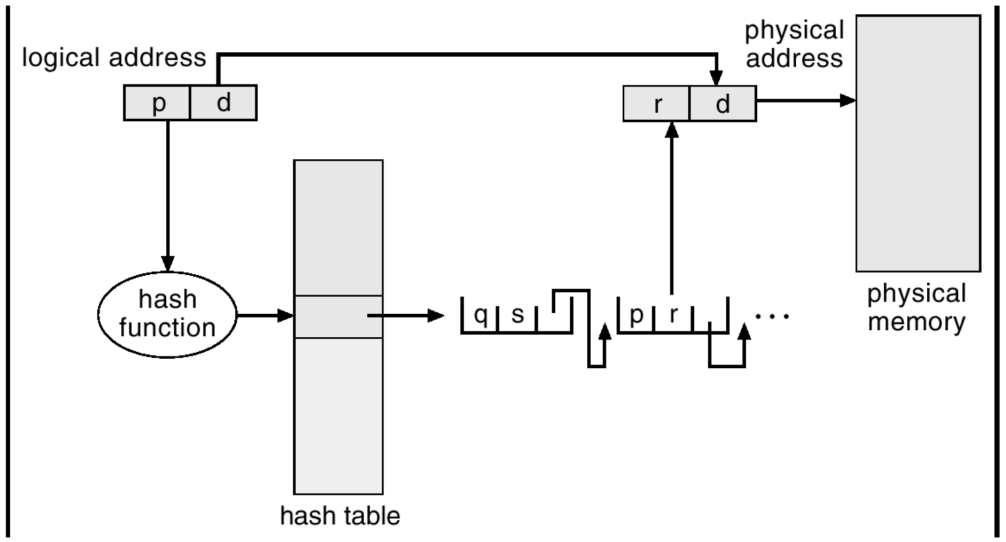

Figure 3: Hashed page tables

5.2 Inverted page tables

These tend to be used when we have a very small amount of memory:

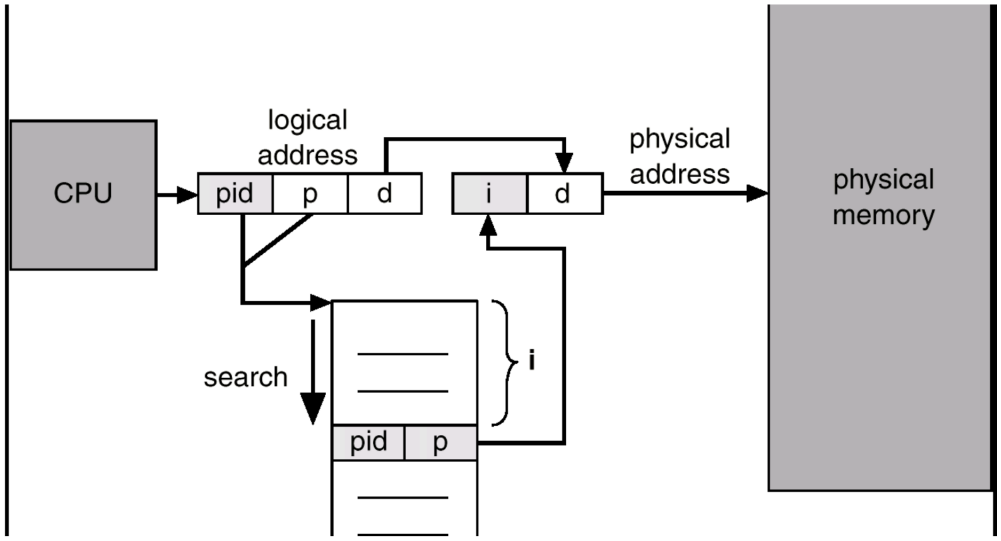

Figure 4: Hashed page tables