1 Virtual memory

Translation look aside buffers (**TLBs**) are (usually) located inside the *memory management unit.*

- They *cache* the **most frequently used** page table entries
- They can be searched in **parallel**
- The principle behind TLBs is similar to other types of **caching** in operating systems
- **Remember**: locality states that processes make a large number of references to a small number of pages

2 Page tables

A **normal** page tables size is proportional to the **number of pages** in the virtual address space, this can be prohibitive for modern machines.
An **inverted** page tables size is proportional to the size of **main memory** (RAM)

- The inverted table contains **one** entry for every **frame** (i.e. not for every page), and it indexes entries by **frame number**, not by page number.
- When a process references a page, the OS must search the (**entire**) *inverted page* table for the corresponding entry (i.e. page and process id), this could be too slow.
- Solution: Use a *hash function* that **transforms** page numbers (n bits) into frame numbers (m bits) - Remember: $n > m$.

2.1 Inverted page tables

- The frame number will be the index of the inverted page table.
- **Process Identifier** (PID) - The process that owns this page.
- **Virtual Page Number** (VPN)
- **Protection bits** (Read/Write/Execution)
- **Chaining Pointer** - This field points toward the next frame that has exactly **the same VPN**. We need this to **solve collisions**.

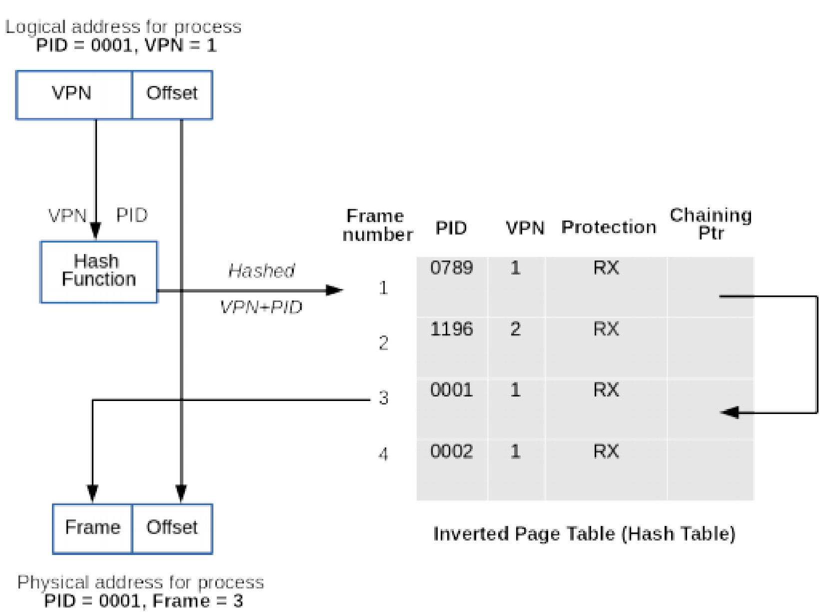

Figure: Address Translation with an Inverted Page Table

Advantages:

- The OS maintains a **single** inverted page table for all processes
- It **saves** lots of space(especially when the virtual address space is much larger than the physical memory)

Disadvantages:

- Virtual-to-physical translation becomes much **harder/slower**.
- Hash tables eliminates the need of searching the whole inverted table, but we have to **handle collisions** (that will also slow down the translation).

TLBs are **particularly necessary** to improve their performance.

2.2 Page loading and replacing

Two key decisions have to be made using virtual memory.

- What pages are **loaded** and when -> predictions can be made
- What pages are **removed** from memory and when -> page replacement algorithms

Pages are **shuttled** between primary and secondary memory

2.3 (On)demand paging

Demand paging **starts** the process with **no pages** in memory:

- The first instruction will immediately cause a *page fault*
- More page faults will follow, but they will **stabilise** over time until moving to the next *locality*.
- The set of pages that is currently being used is called its *working set* (resident set)
- Pages are only loaded **when needed**, i.e. following *page faults*

2.4 Predictive paging

When the process is started, **all pages** expected to be used (i.e. the working set) could be brought into memory **at once**.

- This can drastically **reduce** the page fault rate
- Retrieving multiple (contiguously stored) pages **reduces** transfer times (seek time, rotational latency, etc.)
- Pre-paging loads pages (**as much as possible**) before page faults are generated (a similar mechanism is used when processes are swapped out/in)

2.5 Implementation details

Avoiding unnecessary pages and page replacement is important!. Let ma, p and pft denote **the memory access time** (2 times for single-level page tables) (ranging from 10 to 200ns), page fault rate, and page fault time, respectively, the effective access time is then given by:

$$T_a = (1 - p) * ma + pft * p$$

Note that we are **not** considering **TLBs** here.
The **expected/effective** access time is **proportional** to *page fault* rate when keeping page faults into account. **Ideally**, all pages would have to be loaded **without** demand paging

3 Page replacement

3.1 Concepts

The OS must choose a page to **remove** when a new one is loaded (and all are occupied). This choice is made by **page replacement algorithms** and takes into account:

- When the page is last **used/expected** to be used again
- Whether the page **has been modified** (only modified pages need to be written)
- Replacement choices have to be made **intelligently** (random) to save time/avoid thrashing

Alogrithms:

- **Optimal** page replacement
  When a page needs to be swapped in, the operating system swaps out the page **whose next** use will occur farthest in the future. For example, a page that is not going to be used for the next 6 seconds will be swapped out over a page that is going to be used within the next 0.4 seconds. It **cannot** be implemented in general OS.
- **FIFO** page replacement
  The **simplest** low-overhead page-replacing algorithm that requires little bookkeeping on the part of the operating system. The OS **keeps track** of all the pages in memory in a queue, with the **most recent** arrival at the back, and the oldest arrival in front. When a page needs to be replaced, the page **at the front** of the queue (the oldest page) is selected. While FIFO is cheap and intuitive, it performs **poorly** in practical application
- **Second chance**
  A modified form of the FIFO, works by looking **at the front** of the queue. It checks to see if its **referenced** bit is set. If it is not set, the page is **swapped out**. Otherwise, the referenced bit is **cleared**, the page is inserted **at the back** of the queue (as if it were a new page) and this process is repeated.
- **Clock replacement** Performs the same general function as Second-Chance. Keeps a circular list of pages in memory, with the "hand" (**iterator**) pointing to the last examined page frame in the list. When a page fault occurs and no empty frames exist, then the R (**referenced**) bit is inspected at the **hand's** location. If R is `0`, the new page is put in place of the page the "hand" points to. Otherwise, the R bit is cleared, then the clock hand is incremented and the process is repeated until a page is replaced.
- Not recently used (**NRU**)
  Favours **keeping pages** in memory that have been **recently** used. This algorithm works on the following principle: when a page is referenced, a referenced bit is set for that page, marking it as referenced. Similarly, when a page is modified (written to), a modified bit is set. At a certain fixed time interval, a timer **interrupt triggers** and **clears** the referenced bit of all the pages, so only pages referenced within the current timer interval are marked with a referenced bit.
- Least recently used (**LRU**) Works on the idea that pages that have been **most heavily used** in the past few instructions are most **likely** to be used heavily in the next few instructions too. While LRU can provide **near-optimal** performance in theory (almost as good as adaptive replacement cache), it is **rather expensive** to implement in practice.

Reference section

placeholder