CI583: Data Structures and Operating Systems
Memory management part 2

Page tables

The system we have described so far is a complete page table.

In this system, every page in the addressable memory has an entry in the page table, even though most of these entries will have the validity bit set to zero.

Rather than simply holding an entire page table that maps to all addressable memory, different schemes are used to store page tables more efficiently.

Page tables

Such schemes are needed because page tables take up too much space and have other benefits – for instance, by using an indirect mapping we can increase the addressable memory.

In various ways, these schemes aim to hold in memory *only those parts of the page table that map to the parts of primary storage currently being used*.

Forward-mapped page tables

Forward-mapped (or multilevel) page tables break a virtual memory location down into three parts:

- the Level 1 page number (L1),
- Level 2 page number (L2), and
- the offset.

Forward-mapped page tables

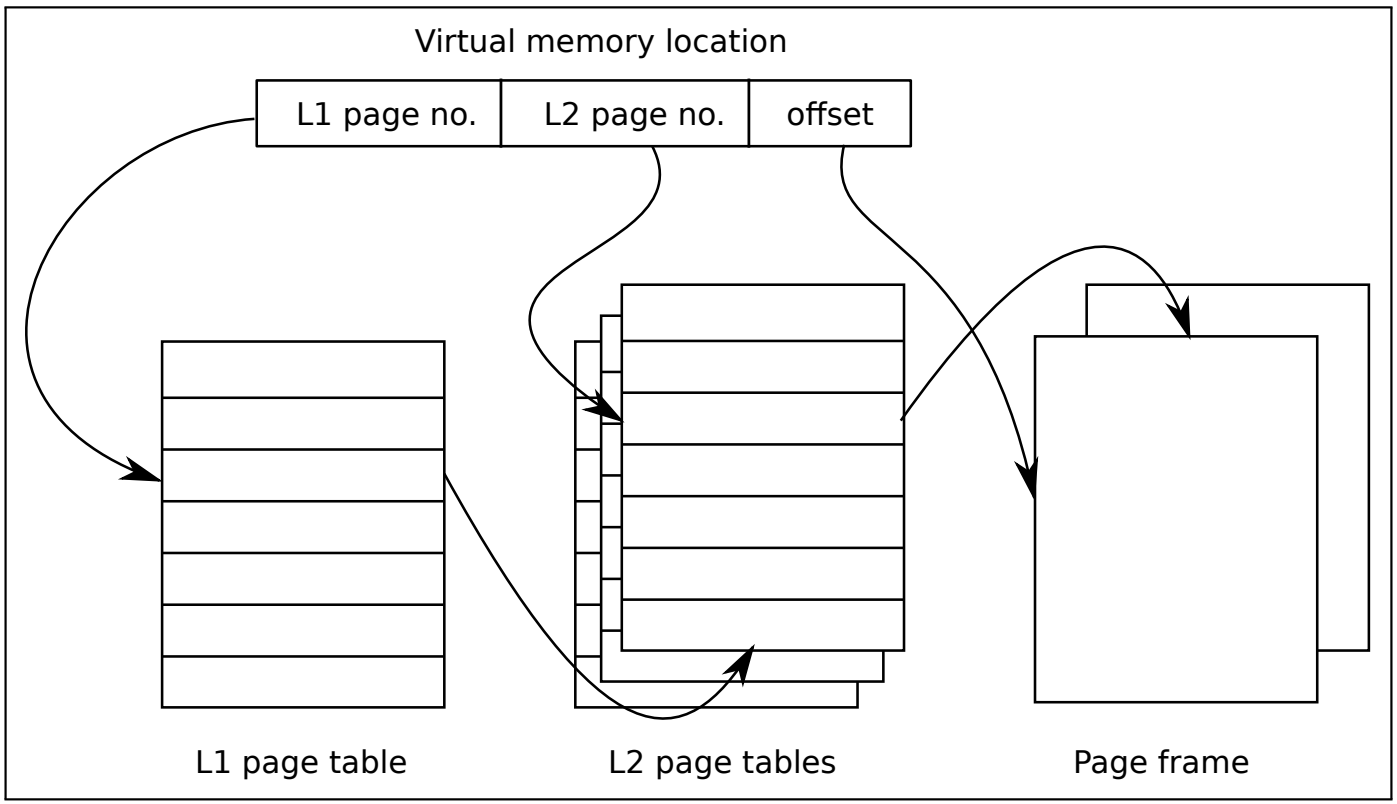

Image adapted from (Doeppner, 2011)

Forward-mapped page tables

As before, entries in the L1 and L2 tables can be valid or invalid.

Looking up a location means finding the entry in L1, which indexes a particular L2 page table.

Then the entry in the L2 table is looked up – this points to a page frame (or is invalid).

Forward-mapped page tables

The benefit of this system is that not all L2 tables need to be in memory at any one time, greatly reducing the amount of memory taken up.

However, translating a virtual memory location now takes two or three lookups.

Linear page tables

Linear page tables break up the entire address space into several (e.g 4) spaces, each with its own page table which is itself held in virtual memory.

Thus, to look up a virtual memory location, we identify the correct space, then look up the page number in the corresponding page table, then finally retrieve the location of the page frame and use the offset to get the right location.

This usually requires fewer memory accesses than forward-mapped tables, because it takes advantage of the fact that the most common pattern of memory access within processes is contiguous.

Hashed page tables

Hashed page tables take a different approach to the ones we've seen so far.

The page table is loaded with translation information for *only those parts of the address space which are in use* (i.e. no invalid entries).

We access the page table using a function which hashes the virtual memory location.

Hashed page tables

Each entry in the page table contains, as well as the page frame number, the original page number and a link to the next entry with the same hash value.

Thus, looking up a page table entry might mean looking up the entry based on the hash, then following one or more links in the table to find the real entry.

Hashed page tables

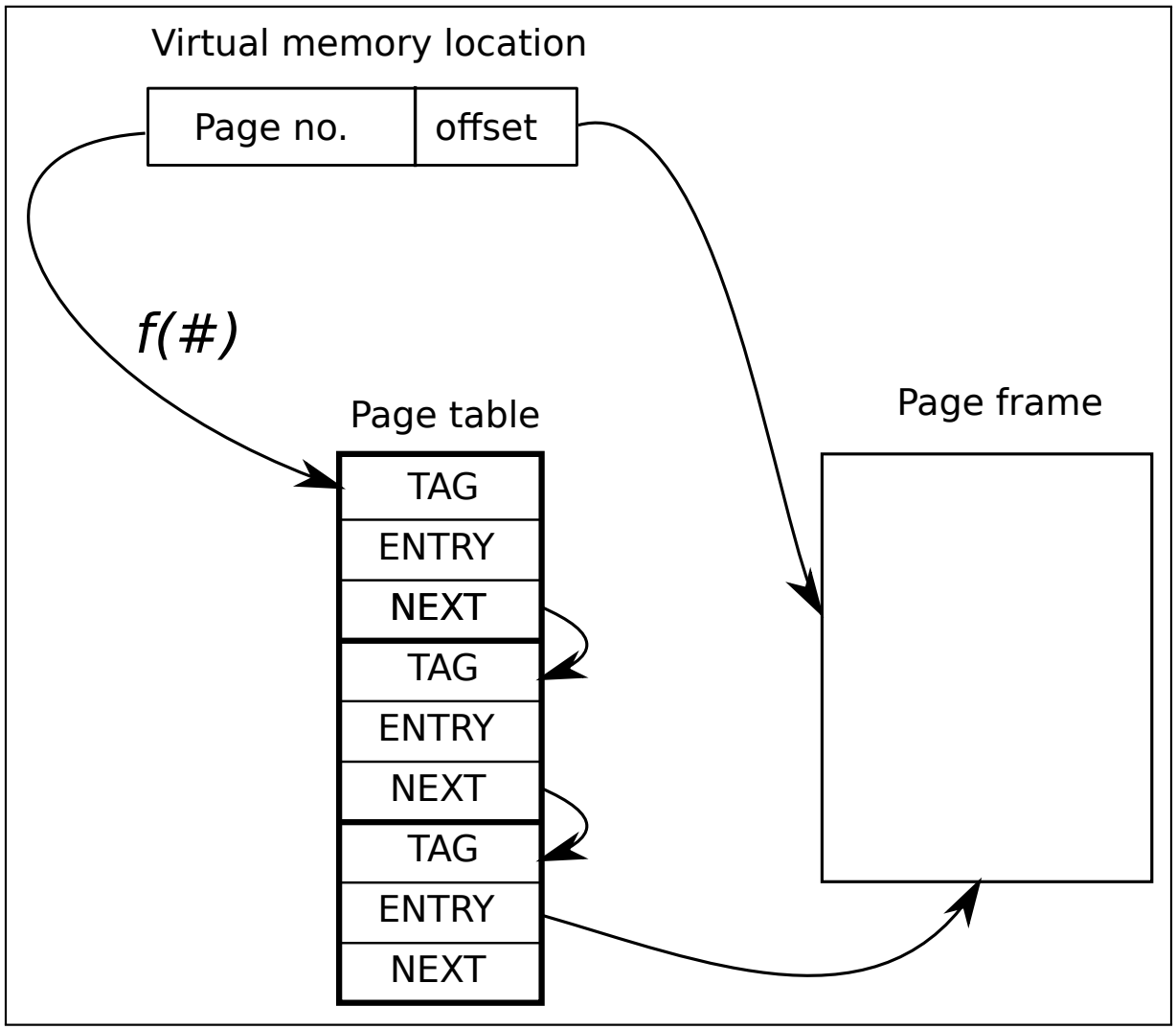

Image adapted from (Doeppner, 2011)

Hashed page tables

This approach is particularly efficient when dealing with small regions of allocated space in a larger, sparsely-populated data region.

However, the page tables become vary large, since each entry requires three words. This problem is solved by using clustered page tables.

Using this approach, multiple pages are grouped together into superpages. Which page we actually require can be inferred from the result of the hash function.

Hashed page tables

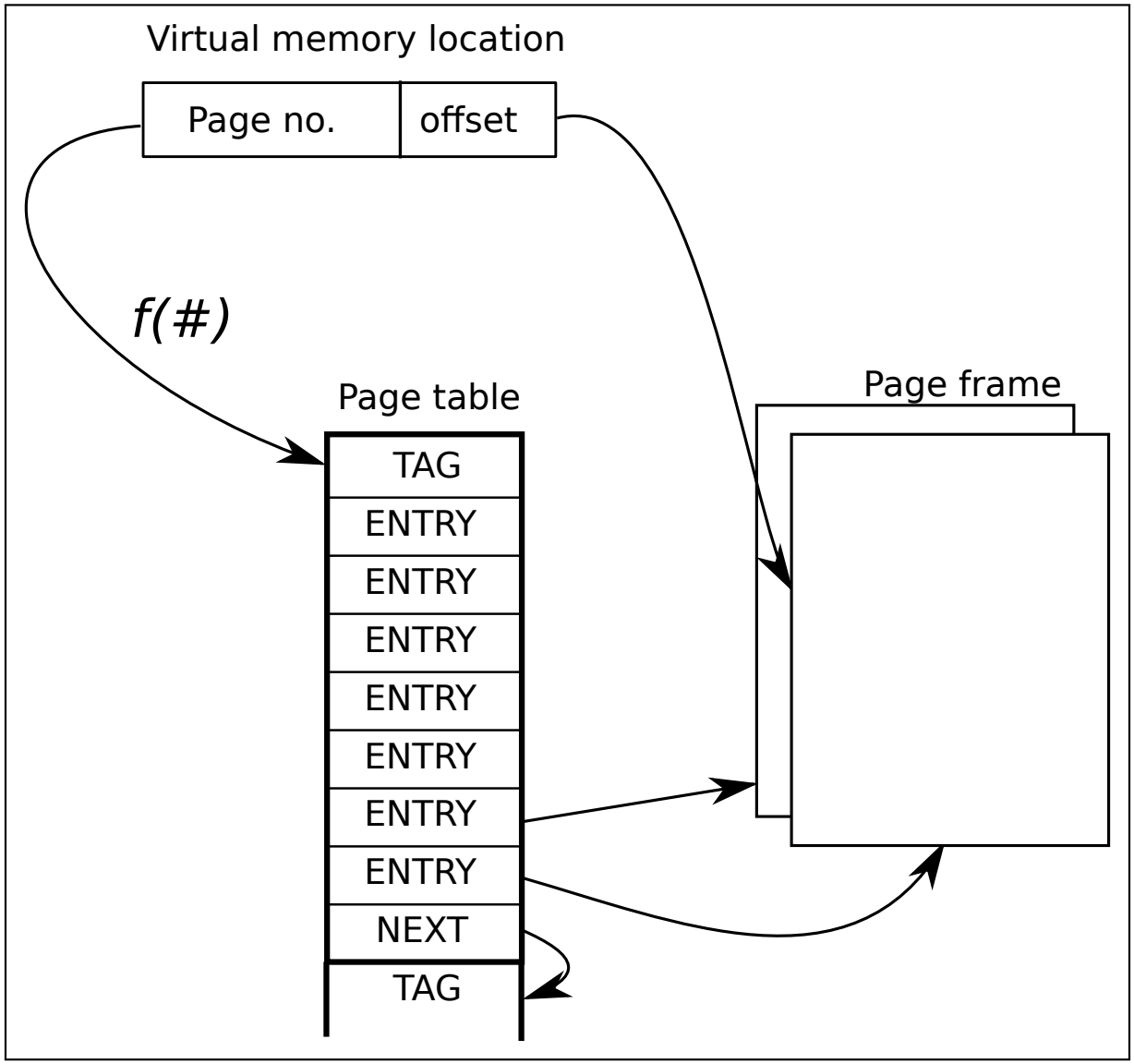

Image adapted from (Doeppner, 2011)

Page tables in 64-bit systems

64-bit systems provide 2^{64} bits of addressable space.

A complete page table would occupy 16 petabytes (as opposed to 4MB for a 32-bit system)!

Page tables in 64-bit systems

The other 32-bit solutions are also "worse" in this setting – e.g. a forward-mapped page table with 4KB pages would require 4 to 7 lookups per translation.

The x64 architecture uses forward-mapped page tables with four levels of 4KB pages and a subsequent 3 levels of 2MB pages.