Page Tables, Caches and TLBs

Page Tables

Suppose that we have a two-level page translation scheme with 4K-byte pages and 4-byte page table entries. What is the format of a 32-bit virtual address? Roughly sketch out the format of a complete page table.

4 kilobytes = 2^12 bytes, which means we need 12 bits to address a specific byte within the page. This leaves us with 20 bits we need to allocate.

Since a page is 4 kilobytes and a page table entry is 4 bytes, this means that we can fit 1024 (2^10) page table entries on a page. This means that we can use 10 bits to represent each level in the page table.

Thus, the breakdown is as follows:

10 bits to reference the correct page table entry in the first level.
10 bits to reference the correct page table entry in the second level.
12 bits to reference the correct byte on the physical page.

Suppose we have a memory system with 32-bit virtual addresses and 4 KB pages. If the page table is full, show that a 20-level page table consumes approximately twice the space of a single level page table.
4 kilobytes = 2^12 bytes, which means we need 12 bits to address a specific byte within the page. This leaves us with 20 bits we need to allocate.

We have 20 bits to work with and a 20-level page table, meaning each level consists of two entries (where a '0' bit references the first entry while a '1' bit references the second entry). Thus there one-bit in the virtual address for each level of the 20-level page table.

The number of total "tables" in this implementation is therefore:
2^0 + 2^1 + ... + 2^19 = 2^20 - 1

Since each table has two entries (one corresponding to '0', one corresponding to '1'), this means that we have a total of 2^21 - 2 entries.

A single-level page table on the other hand, has 2^20 entries, meaning that there are a total of 2^20 entries in such an implementation. Therefore the 20-level page table consumes approximately twice the space when full.

Show that the above is not necessarily true for a sparse page table.
Imagine if we only have 1 page of physical memory allocated to a virtual space. This means that we have only 20 tables, since we just need one entry per level of indirection. Each table has 2 entries, so we have a total of 40 entries for our page table implementation.

In the single-level case, in the sparse memory scenario, we still have 2^20 entries allocated, meaning that the 20-level page table is theoretically good for sparse memory usage (although 20 memory accesses to determine a translation is still expensive and therefore not practical).

Caching

What are the two principles of locality that make implementing a caching system worthwhile?
Spatial Locality - If I access a block of data, I am likely to access blocks of data next to it in the future. A simple example is sequentially reading through an array. In this scenario, a caching system with a large block size takes

advantage of spatial locality because when I read the first element, I read in seven additional elements so that the next seven reads are from the cache and therefore fast.

Temporal Locality - If I have accessed a block of data before, I am likely to access this block of data again. A simple example is if I have a piece of code that constantly references/updates a global counter.

List and (briefly) describe three sources of cache misses.
Compulsory Miss (Cold Start) - The very first time I load a program, I am going to get a cache miss since I have to load it in. The code had to be put into the cache at some point.

Collision Miss - If I am using a direct-mapped cache (where the index is determined by % 4), then a block from memory address x will be placed into the same index as a block from memory address x + 4. This means that I am forced to evict the old entry (x) in favor of (x + 4), even if my cache is not yet full. Fully-associative caches do not suffer from this problem, but they are more expensive to implement because I have to compare with all the entries in the cache if I am trying to determine whether a block of data exists in it.

Capacity Miss - The cache is full and to store another entry in it, I have to evict a previous entry.

Coherence Miss - I have a value 'x' in the cache, but the value of 'x' in memory has been updated, meaning that the value stored in my cache is now out of date. This can occur with multiple CPUs or Direct Memory Access (DMA).

True or False: When looking up an entry in the cache, it is not sufficient to find a matching index.
True. Look at the explanation for Collision Miss: Two different addresses (x, x + 4) hash to the same index. Therefore if I am looking for x and I see an entry in the cache, that does not mean that the entry corresponds to x - it could correspond to 'x + 4.' You need to check the "tag" (which contains the rest of the bits of the virtual address) to make sure that the entry located at the index is the one you really want.

What are the advantages / disadvantages of using a Fully Associative Cache?
Advantage: Collision hits are non-existent since a cache entry can be placed anywhere.

Disadvantage: More expensive to implement because to search for an entry we have to search the entire cache. This search is done in parallel, but this requires a lot of extra hardware.

Briefly explain the difference between a write-through and a write-back cache.
A write-through cache will write the value back to memory when it changes. For example if the value 'x' is stored in the cache and I increment it by one, then the incremented value is written to the cache and to physical memory as well.

A write-back cache will only write the value back to memory when the entry is evicted from the cache (either due to a Collision or a Compulsory miss).

Translation Lookaside Buffers (TLBs)

True or False: TLBs are organized as a directly-mapped cache to maximize efficiency.
False: A TLB miss is costly so we want to reduce the chance of one. We can do this by using a fully-associative cache, which eliminates the possibility of a Collision miss.

True or False: The TLB in Nachos needs to always be invalidated on every context-switch.
True: Otherwise a different process ID could obtain a translation to a different process' virtual page number (and therefore physical memory that does not belong to it).

What are the disadvantages of using the higher order bits to index into a TLB?
*For very small programs, all the memory addresses would hash to the same index in the TLB, creating *a lot* of collisions.*

What are the disadvantages of using the lower order bits to index into a TLB?
The first page of the code, stack and heap could map to the same index in the TLB, causing a lot of collisions (initially)

*Moral of the story: TLBs use *some* level of associativity to avoid collisions in the above two scenarios.*

What is the effective access time for TLB with 80% hit rate, 20ns TLB access time and 100 ns Memory access time (assume two-level page table that is not in L2 cache)?
$0.8(20) + 0.2(200 + 20) = 16 + 44 = 60$ ns

The first term: There is an 80% chance we only need to spend 20ns to obtain the translation.

The second term: In the 20% chance we get a TLB miss, we need to traverse two pointers to get the translation and then populate the entry in the TLB.