I began this lecture with a high level overview of some of the main concepts we have covered concerning memory hierarchies. See the slides for a list.

The remainder of the lecture consisted of two parts. The first part showed another way of organizing page tables. When I originally introduced page tables in lecture 17 and briefly discussed how to organize them in memory, I only mentioned one possibility which was to put the page tables in the part of the kernel that is not itself paged. In this lecture I tell you another way to do it. This is a relatively advanced topic and not generally covered COMP 273, so feel free to skip to part 2 which is more core.

In part 2 of the lecture I will say more caches, in particular what happens when there is a cache miss.

**Part 1: How to put page tables in paged memory (Advanced Topic)**

Let’s first consider how big is a page table. Assume each page is $2^{12}$ bytes. If we have a 32 bit virtual address space, then this gives $2^{20}$ possible pages. A page table therefore has $2^{20}$ entries. Each entry contains:

- a physical page number (which can be either a main memory page number or a hard disk page number)
- a valid bit that says whether this page is in main memory (1) or on the hard disk (0)

[ASIDE: Here we are assuming for simplicity that that all pages in the virtual memory address space will have corresponding physical pages. In reality, this will not be true, for example, there may be a large gap between the heap and stack which is empty. In that case, you could have two valid bits instead of 1. The first would say whether there is a page anywhere in physical memory, and the second could say whether the page (if it exists) is in main memory or on the hard disk. For this course, we just assume the simple scheme in which there is one valid bit.]

In our earlier example, we had a 1 TB hard disk ($2^{40}$ bytes) which has $2^{28}$ pages, so we would need 28 page bits + the control bits in our page table entry. Let’s just say that each page table entry has 1 word (32 bits). So, the page table would be 4 MB, namely $2^{20}$ entries with 4 bytes each.

In lecture 17, I mentioned one method for storing the page table would be to put it in a non-paged part of main memory. Here I introduce an alternative way to store the page table, which is to put it in paged memory. Suppose we partition a page table itself into $2^{10}$ pages, i.e. $2^{22}/2^{12} = \text{page table size} / \text{bytes per page}$. To keep track of where these $2^{10}$ (page table) pages are, we use another table, which has $2^{10}$ entries, each one word (namely the physical address of a one page chunk of the page table). This special page is called the page table directory, ... or if you like, the “page-table table”.

To visualize the above, imagine a tree whose root node is a page, namely the page table directory. (See the figures in the slides!) It contains $2^{10}$ physical page numbers. Each of these entries says where a one page chunk of the page table is. Each one page chunk may be in main memory, or it

---

1. I also briefly mentioned – see bottom of p. 5 of lecture 17 – that one typically does not store all $2^{20}$ possible page table entries. Rather one stores only those entries for which there is an actual page present – either in main memory or on the hard disk. This is related to the ASIDE above, e.g. that there may be many unused pages between the heap and stack and you don’t really need to represent these pages in the page table since they don’t correspond to any virtual to physical translation.
may be on the hard disk. You can assume that the only one page chunks of the page table that would be on the hard disk would be those chunks for which there are no pages in physical memory (either in main memory or on disk), that is, the gaps between the heap and stack. Note that it would be very inefficient otherwise, i.e. to have parts of the page table on the hard disk since you would generate a page fault just to access the page table!

With the above scheme, a virtual address (32 bits) is partitioned into three components:

- a 10-bit index to a one page chunk of the page table: this is an index in the page table directory; at that entry you will find the physical page number of this one page chunk of the page table.

- a 10-bit offset that is used to index in this one page chunk of the page table; at that entry you will find a physical page number of page you are looking for.

- a 12-bit offset that is used to index into the physical page you have found, and gives you the byte you are looking for, namely the physical byte that corresponds to the byte indexed by the 32 bit virtual memory address.

Keep in mind that there is one page table for each process. The kernel needs to keep track of memory usage and page tables for each process, and it needs to use this information when it decides which pages to swap during a page fault. i.e. if no pages are available in main memory, then the kernel needs to make room for a new page – by moving a page from main memory to the hard disk and copying the desired page from hard disk to main memory. This is a topic that is covered in an Operating Systems course (COMP 310), but let me say a bit more about it here in case you are interested.

How does the kernel decide which page to move from main memory to the hard disk? One common policy is to move the page that was least recently used. To do this, the kernel needs to keep track of when each page was used, and maintain a sorted list. The move-to-front algorithm comes up here. Whenever a page is used, it is moved to the front of the list (most recently used). When some page must be removed from main memory, the kernel chooses the page at the back of the list (least recently used). Over time, if a main memory page is not used, it drifts further and further back in the list, as pages that are used are moved to the front. **You will learn about this in COMP 310.**

**Part 2: TLB/cache: hit and miss**

Last lecture, we looked at caches. We considered direct mapping methods and associative mapping methods. The emphasis in the last lecture was on indexing methods, and how we read from a cache assuming that the cache has the word we word we want. This lecture, we will consider two other aspects of caches. The first is what happens when the address we are indexing is not in the cache. This is called a miss. The second aspect is how we write to a cache, either from a register to the cache (in the case of a store word), or when we copy a block from main memory to the cache in the case of a miss. **Today we will consider only direct mapped caches. (The arguments for associative caches are similar, but more detailed and complicated.)**
TLB miss and TLB refill (and page fault)

We first deal with the TLB, which is a cache for the page table. Consider what happens if the TLB doesn’t contain a desired virtual-to-physical translation. This is called a TLB miss. If a TLB miss occurs in your program, an exception results. Your program jumps to the exception handler which analyzes what the exception was, and then jumps to a special kernel program which handles TLB misses – namely the TLB miss handler. This kernel program consults the page table of the current process, to see if the desired word is in memory (i.e. if the page table entry’s valid bit is 1):

If the page valid bit is 1, then the TLB miss handler fills the appropriate entry in the TLB (called a TLB refill), and sets the valid bit for that TLB entry to 1. The kernel then returns control to the program so that it can continue its execution, namely it can perform the virtual-to-physical translation that had caused the TLB miss.

If, however, the page valid bit is 0, then the address that the program wants is not in main memory. Rather it is on the hard disk (or DVD,...). A page fault occurs. The TLB miss handler now calls the page fault handler. The page fault handler arranges for the desired page to be copied from the hard disk to main memory. (We’ll see how, in an upcoming lecture.). The page fault handler then updates the page table appropriately, for example, it changes the page valid bits and physical page numbers of the swapped pages. When the page swap is completed, the page fault handler returns to the TLB miss handler. Now, the requested page is in main memory and the page valid bit in the page table is 1. So the TLB miss handler can copy the page table entry into the TLB. The TLB miss handler then returns control to the original process.

Let us next turn to the instruction and data cache. In many architectures (such as MIPS), there is a one cache for the instructions and another cache for the data. The instruction and data caches have a subtle difference: instructions are only fetched (read) from memory, but data can be read from or written to memory. For the instruction cache, blocks are copied from main memory to the cache. For the data cache, blocks can be copied either from main memory to the cache, or vice-versa. There can also be writes from registers to the data cache e.g. by sw (or swc1) instructions.

Instruction Cache

Let’s begin with the instruction cache, which is simpler. Suppose we are fetching and the instruction is in the cache, then we have a hit. This was the case discussed last lecture. If, however, the instruction is not in the cache, we have a miss. This causes an exception – a branch to the exception handler which then branches to the cache miss handler. This kernel program arranges for the appropriate block in main memory (the one that contains the instruction) to be brought into the cache. The valid bit is then set, indicated that the block now in the cache indeed represents a valid block in main memory. Note that the block must be in main memory rather than on the hard disk since we only get to this stage once we have passed the TLB stage, which ensured that the page was in main memory. (See above.)

Now the cache miss handler returns to the process and the instruction is fetched (a hit).

Data Cache - write-through policy

The data cache is more complicated. Since we can write to the data cache (sw), it can easily happen that a cache line does not have the same data as the corresponding block in main memory. There are two policies for dealing with this issue: “write-through” and “write-back”. The write-through...
policy which ensures that the cache block is consistent with its corresponding main memory block. We describe it first.

Consider reading from the data cache (as in \texttt{lw}). If the word is in the cache, then we have a hit. This was covered last lecture. If there is a miss, however, then an exception occurs and we need to replace the cache line. The cache miss handler copies the appropriate block from main memory to the cache (and fills the various fields of the cache line -- see last lecture). The previous line of the cache is erased in the process. Although that line is erased, however, this is no problem for the write-through policy since, as we see next, it ensures that the cache line (just erased) has the same data as its corresponding block in main memory, and so the data is not lost.

Consider happens when we write a word from a register to the cache (\texttt{sw} or \texttt{swc1}). First suppose that there is a hit -- the cache has the correct line. The write-through policy is to copy the word from the register to the cache line and also to copy the word back to the appropriate block in main memory, so that main memory is consistent with the cache lines.

If the word is not in the cache, then an exception occurs -- a cache miss. The cache miss handler arranges that the appropriate block is transferred from main memory to the cache. The handler then returns control to the program which tries to write again (and succeeds this time -- a hit).

To summarize ("write through"):

<table>
<thead>
<tr>
<th>hit</th>
<th>miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>copy block main mem → cache (and set valid bit)</td>
</tr>
<tr>
<td></td>
<td>copy word reg → cache</td>
</tr>
<tr>
<td>write</td>
<td>copy block main memory → cache (and set valid bit)</td>
</tr>
<tr>
<td></td>
<td>copy word register → cache</td>
</tr>
<tr>
<td></td>
<td>copy word cache → main mem</td>
</tr>
</tbody>
</table>

Data cache: "write back" policy

The second policy avoids copying the updated cache block back to main memory unless it is absolutely necessary. Instead, by design, each entry in the cache holds the most recent version of a block in main memory. The processor can write to and read from a block in the cache as many times as it likes without updating main memory -- as long as there are hits. The only special care that must be taken is when there is a cache miss. In this case, the entry in the cache must be replaced by a new block, but before the new block can be read into the cache, the old block must be written back into memory so that inconsistencies between the block in the cache (more recent version) and the block in main memory (older version) are not lost. This is how the "write back" scheme delays the writing of the block to memory until it is really necessary.

To keep track of which lines in the cache are consistent with their corresponding blocks in main memory, a dirty bit is used -- one per cache line. When a block is first brought into the cache, the dirty bit is set to 0. When a word is written from a register to a cache block (e.g. \texttt{sw}), the dirty bit is set to 1 to indicate that at least one word in the block no longer corresponds to the word in main memory.

Later, when a cache miss occurs at the cache line, and when the dirty bit is 1, the data block at that cache line needs to be written back to main memory, before the new (desired) block can be brought into the cache. That is, we "write-back". This policy only writes a block back to main memory when it is really necessary, i.e. when the block needs to be replaced by another block.
Note that there is just one dirty bit for the whole block. This bit doesn’t indicate which word(s) is dirty. So the whole block is written back to main memory and a whole new block is brought in.

This “write-back” policy helps performance if there are several writes to individual words in a block in the cache before that block is replaced by another. For example, if you have an array (which is too big to put into registers) which you access frequently, both reading and writing, then it would good to keep it in the cache as long as you can.

The following table summarizes the steps of the data cache write-back policy. Note that the misses take more time, both for (data) cache read and write.

<table>
<thead>
<tr>
<th></th>
<th>hit</th>
<th>miss</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>read</strong></td>
<td>copy word: cache → reg</td>
<td>copy block: cache → main mem (<em>only if valid and dirty bits are 1</em>)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>copy block: main memory → cache</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and set dirty bit = 0, valid bit = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>copy word: cache → register</td>
</tr>
<tr>
<td><strong>write</strong></td>
<td>copy word: reg → cache</td>
<td>copy block: cache → main mem (<em>only if valid and dirty bits are 1</em>)</td>
</tr>
<tr>
<td></td>
<td>(and set dirty bit = 1)</td>
<td>copy block: main mem → cache (and set valid bit = 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>copy word: reg → cache (and set dirty bit = 1)</td>
</tr>
</tbody>
</table>

Notice that in the write back scheme, a “write hit” is cheaper and a “read miss” is more expensive. For large caches, the hit rate is typically over 95 per cent. For this reason, a write back policy tends to give better performance than the write through policy for large caches.

Finally, in the above, I discussed only the situation for direct mapped TLB and caches. If we use an associative mapped cache, then there is some choice about which cache line gets replaced when there is a miss. A common policy is to replace the least recently used one. How this is done (with circuits) is beyond the scope of the discussion. I mention it only because the LRU policy comes up in Assignment 3 Question 3 and so you need to know what it is.