CSO: Recitation 13

Goktug Saatcioglu

12.05.2019

1 In-Class Exercise

Let's practice both address translation and caching. Assume we have an 8-bit system. What does this mean? It means that we have 2^8 possible memory locations available for use. In other words, we have $2^8 = 256$ bytes of physical memory to use. In this case we have a system with 8-bit wide physical addresses. Furthermore, assume the width of virtual addresses is 8-bit. Consider the following two scenarios

1. Assume we wish to split the virtual address space into 16 byte pages. How many page table entries do we require? Well the virtual address space is $2^8 = 256$ bytes so if we split it into 16 byte pages then we will have 16 pages as $\frac{256}{16} = \frac{2^8}{2^4} = 2^4 = 16$. In this case, what should be the offset then? Well we have 16 pages and each page has 16 entries so we for any page we need to accomodate for 16 entries. We know that $16 = 2^4$ so it must be that we use 4 bits for the offset. Notice that we could have also answered another simple question. How many bits do we require for the VPN (virtual page number)? Well the page table has 16 entries and $2^4 = 16$ so 4 bits works. There is complementarity here so you may approach the problem of decomposing the virtual address into its component from either question. Either way you will see that the first (most significant) 4 bits of the VA is for the VPN and the rest of the 4 bits are for the offset.

2. Assume we wish to split the virual address space again into 16 byte pages. However, we wish for the level 0 page table to have only 8 entries (due to some restrictions). So we require another level, say level 1, for the whole page table. How many entries should be in the 1st level page table? Well we only have 8 in the first level so the second level must have 2 each so that we again split the address space into 16 chunks of 16 byte pages. Now we ask, how many bits do we need for the first level of the page table? Well there are 8 page table entries so 3 bits should suffice. Then each second level page table has 2 entries so we only need 1 bit for the second level. So VPN0 is 3 bits and VPN1 is 1 bits. Again, we require 8 bits for the offset as we are still splitting the address space into 8 chunks of 16 bytes. Notice how the offsets do not change between scenario 1 and 2. Can you explain why? Make sure you understand it very well. The only difference here is how we are organizing our page tables.

We will use scenario 2 for our page table design. How large should the PPN be? For the 0 level it can 8 bits as the other page tables are stored in some other area of the memory and we said that the memory addreses are 8 bits. For the first level PPN we have 4 bits for the offset so PPN1 can be 4 bits. Assume that the page table at some instance of the system looks like this.

level 0

VPN0	PPN0	V
00	A4	0
01	FE	1
02	18	1
03	--	0
04	BB	1
05	--	0
06	--	0
07	D6	0

level 1: A4

VPN1	PPN1	V

0	3	1
1	1	1

level 1: FE

VPN1	PPN1	V
0	7	1
1	0	0

level 1: 18

VPN1	PPN1	V
0	5	1
1	4	0

level 1: BB

VPN1	PPN1	V
0	3	0
1	7	1

level 1: D6

VPN1	PPN1	V
0	1	0
1	1	0

For each of the virtual addresses given below, calculate the the VPN and the page table offset. Furthermore, outline the steps we take as we look up an address in the page table and whether there was a page hit or miss. If there is an hit than indicate what the resulting address is. Otherwise, explain why we had a page miss.

1. `1A`: We see that VPN0 is `0` and VPN1 is `1`. Furthermore, the offset is `A`. We look at the level 0 page tabe and see that while VPN0 does indeed have an entry it is not valid as the validity bit is set to 0. So, we get a page miss.

2. `67`: We see that VPN0 is `6` and VPN1 is `1`. Furthermore, the offset is `7`. We look at the level 0 page table and see that VPN0 is not valid and also has no entry. So, we get a page miss.

3. `46`: We see that VPN0 is `2` and VPN1 is `0`. Furthermore, the offset is `6`. We look at the level 0 page table and see that VPN0 is valid and PPN0 is `18`. We look at the level 1 page table located at memory location `18` and use VPN1, which is `0`, to find the relevant entry. We see that the validity bit of this entry if 0 so this is a page miss.

4. `97`: We see that VPN0 is `4` and VPN1 is `1`. Furthermore, the offset if `7`. We look at the level 0 page table and see that VPN0 is valid and PPN0 is `BB`. We look at the level 1 page table located at memory location `BB` and use VPN1, which is `1`, to find the relevant entry. We see that the validity bit of this entry if 1 so this is a page hit. PPN1 is `7` so the translated address becomes `77`.

What could be problematic with our implementation? Each address translation requires 3 potential address lookups. We wish to lower this amount so we will use a cache to store some of the most recent lookups so that address translation can be done faster. This is in the form of a TLB (translation lookaside buffer). Assume we have a TLB cache that holds 4 sets and is 2-way set associative. Firstly, we have to determine the index, tag and PPN parts of a VA. If we have 4 sets then we can represent 4 sets with 2 bits since $2^2 = 4$.

This means that we need to reserve 2 bits for the index lookup. The block size is still 16 bytes so the offset stays at 4 bits. That leaves 2 bits for the tag. The tag will be the higher order 2 bits, the next two bits will be the set index and the last 4 bits will be the page offset. Assume that the TLB at some instance of the system looks like this.

Idx	Tag	PPN	V	Tag	PPN	V
0	0	--	0	2	3	0
1	2	--	-	F	A	0
2	1	7	1	3	--	0
3	0	5	1	A	E	0

For each of the virtual addresses given below, calculate the page table offset, the TLB set index and the TLB tag. Furthermore, outline the steps we take as we look up an address in the cache and whether there was a cache hit or miss. If there is an hit than indicate what the resulting address is. Otherwise, explain why we had a cache miss.

1. `BA`: The pto is `A`, the TLB set index is `2` and the tag is `3`. We find cache index 2 and then find tag 3 and notice the validity bit is 0. Thus, this is a cache miss.

2. `27`: The pto is `7`, the TLB set index is `0` and the tag is `2`. We find cache index 0 and then find tag 2. We see that while there is a PPN there the validity bit is set to 0 so this is a cache miss.

3. `77`: The pto `7`, the TLB set index is `3` and the tag is `1`. We find cache index 3 and then find tag 1. We see that the validity of this cache location is set to 1 so we retrieve the relevant PPN which is 7. This is a cache hit and the translated address, i.e. PA, is `77`.

The TLB is nice but we still have to access the memory after translating the address. We would like to have something where we can temporarily store values that we grab from the memory. This is the cache! We will be using a 4-way set associative 4-byte cache lines and 4 sets. Assume the memory is byte addressable and memory accesses are to 1-byte words. Again our addresses are 8-bits in width. Since we have four sets the set index can be represented using two bits ($2^2 = 4$). The block offset should express our ability to retrieve any one of the 4 bytes in a cache line. Since $2^2 = 4$ 2 bits should work well. The rest of the bits will be then used for the tag. The tag is the first (higher-order) 4 bits which is followed by 2 bits of cache set and the last 2 bits are the block offset. Assume that the cache at some instance of the system looks like this. Anything not shown below is invalid.

Idx	Tag	Data	V	Tag	Data	V
0	4	--	0	0	5D 4D F7 DA	1
0	6	32 21 1C 2C	0	3	69 C2 8C 74	0
1	7	69 C2 8C 74	1	3	--	0
1	D	ED 32 0A A2	1	E	32 21 1C 2C	1

For each of the addresses given below, calculate the cache tag, the cache set index and the cache block offset. Furthermore, outline the steps we take as we look up some data in the cache and whether there was a cache hit or miss. If there is an hit than indicate what the resulting data is. Otherwise, explain why we had a cache miss.

1. `44`: The tag is `4`, the index is `1` and block offset is `0`. Looking at the validity bit in that location reveals that this part of the cache is not valid and we get a cache miss. Furthermore, there is no data there anyways.

2. `31`: The tag is `3`, the index is `0` and block offset is `1`. Looking at the validity bit in that location reveals that this part of the cache is not valid even though there is something there. So, we get a cache miss.

3. `D7`: the tag is `D`, the index is `1` and block offset is `3`. Looking at the validity bit in that location reveals that this part of the cache is valid. So, we grab the data `ED 32 0A A2` and since the block offset is `3` we want the last byte. Thus, the cache return `A2`.

As a bonus question, calculate the address of the second byte of the data `69 C2 8C 74` on the second line. The tag is easy as it is clearly `7`. The index should be `1` and byte offset should be `1` so the address should be `76`. We can further optimize our address translation and caching scheme (see 3.6 for more details).

2 Virtual Memory

2.1 Isolation

When we have many applications running on our computers we would like to have it such that if one of them crash then the whole system does not crash. That is, we would like to have isolation of processes. A process is the unit of isolation and we define isolation as the enforced seperation of processes to contain the effects of failures.

2.2 Process

- An instance of a computer program that is being executed
- Program vs. Process
  - Program: a passive collection of instructions
  - Process: the actual execution of those instructions
- Different processes have different process id - `getpid()`: function that returns id of current process
- Command `ps`: list all processes

To run a program, OS starts a process and provide services through system calls (`getpid()`, `fopen()`).

2.3 Processes Share the Same Physical Address Space

- The requirements:
  - Different processes use the same address to store their local code/data.
  - One process can not access another process' memory
- Why?
  - Isolation: prevent process X from damaging process Y
  - Security: prevent process X from spying on process Y
  - Simplicity: Systems (OS/Compiler) can handle different processes with the same code. (etc. linking or loading)
- How?
  - Virtual Memory

2.4 Memory Management Unit (MMU)

We use what is referred to as a MMU to translate virtual addresses to physical addresses. Since both the virtual memory space and physical memory space are contiguous we can build this mapping at a coarse granularity. We can split the virtual/physical memory space into contiguous blocks of the same size which we call pages and then create a page table that maps the virtual pages to physical pages. So, if a we have a virtual address then we use the MMU to find the mapping to the real page table we are looking for along with the offset in that page of the data we want. These two components translate into a physical address which we can then use to access the memory. For example, if we have a 64-bit machine then there are 2^{64}

physical addresses we can support. If we did not use the page table then our virtual address space will also be of size 2^{64}. If we split this virtual address space into pages of size 4KB then our virtual address space will become $\frac{2^{64}}{2^{12}} = 2^{52}$ which is a significant reduction.

2.5 Address Translation

- Virtual Address $\mapsto$ Physical Address
- Calculate the virtual page number. This corresponds to the bits 63 to 12 (inclusive) of the virtual address layout. Virtual page number (VPN).
- Locate the data from the according physical page. This corresponds to the bits 11 to 0 (inclusive) of the virtual address layout. Page offset (VPO).
- Memory address width: 64-bits
- Page size: 4KB (2^{12})
- Furthermore, page table entries (PTE) encode permission information. `S` means accessible by OS only, `W` means writeable and `P` means present. So a PTE containts some auxillary information, the physical page # and this permission information.

2.6 Multi-level Page Tables

There is a problem with 1-level page tables. For 64-bit address space and 4KB page size we will require 2^{52} page table entries to translate the virtual address to a physical one. The problem is that we would like to reduce the number of page table entries required. The solution is multi-level page table (for example, x86-64 supports 4-level page table).

2.7 Multi-level Page Tables Details

We have a root address that maps to 0-level page table. Each entry in the 0-level page table maps to a 1-level page table. Each entry in the 2-level page table maps to a 3-level page table and so on. So our virtual address could look like as follows:

```
  -------------------------------------------------------------------------
 | Reserved | L0 Offset | L1 Offset | L2 Offset | L3 Offset | Page Offset |
  -------------------------------------------------------------------------
 ^         ^ ^         ^ ^         ^ ^         ^ ^         ^ ^            ^

63        48 47       39 38       30 29       21 20       12 11           0
```

In the CPU register `CP3` we store the physical address of the first entry at $L0$ (i.e. the 0-level page table). Each offset calculates which entry we would like to access with respect to the base address of that table. So the $L1$ offset gives us the value of base address of $L1$ plus the $L1$ offset. Of course the base address of $L1$ is calculated using the base address of $L0$ and the $L0$ offset. So we need to use the `CP3` register to kick off this whole process. Finally, for each process we realize the same virtual address space by having the OS set up a seperate page table for each process. When executing a process `p`, MMU uses `p`'s page table to do address translation.

2.8 Demand Paging

We have one more issue to consider. Why does multi-level page table reduce page table size? We know that the 4-level page table is not fully occupied. So te OS constructs the page table on demand. This is known as demand paging.

- Memory Allocation (e.g., `p = sbrk(8192)`)
- User program to OS: Declare a virtual address range from `p` to `p + 8192` for use by the current process.
- OS' actions: Allocate the physical page and populate the page table.

A possible run of a page lookup with demand paging look as follows.

1. OS adds $[p, p + 8192)$ to the process' virtual address info.
2. MMU tells OS entry is missing in the page at level L. (Page fault)
3. OS constructs the mapping for the address. (Page fault handler) [The OS will end up allocating new pages in the memory]
4. OS tells the CPU to resume execution.
5. MMU translates address again and accesses the physical memory.

2.9 Segmentation Fault

Address translation can fail due to 2 reasons: (1) MMU reads a missing page table entry (PTE) meaning PTE's present bit is unset or (2) MMU reads a PTE with wrong permission for the access. The latter case can occur when the write bit is unset for a write access or OS bit is unset for a write access. For case (1) and (2) MMU generates "page fault" which is then handled by the OS where for case (1) the OS will fix the problem (e.g. by doing paging) or for case (2) the OS aborts the process with "segmentation fault."

2.10 Memory Access Cost

- Memory access latency: 100 ns or 160 to 200 CPU cycles
- Instructions that do not involve memory access can execute very quickly: Instructions per CPU cycle ¿= 1
- If we have 4 level page table how many memory accesses will we need to access some data with some virtual address? A: 4 page table accesses plus one time data access which is 5 memory accesses
- How can we speedup address translation? A: Translation lookaside buffer (TLB)

2.11 Translation lookaside buffer

The TLB is a small cache in the MMU which maps virtual page numbers to physical page numbers. The general way a TLB works as follows:

1. Calculate VPN (virtual page number)
   (a) `VPN = VA >> 12`
   (b) `Offset = VA & 0xfff`
2. Check TLB
   (a) `Index = VPN % 4` (this is for when the TLB has 4 entries only, in general the index lookup may change)
   (b) Check if `TLB[Index].VPN == VPN`
   (c) TLB hit ==¿ `PA = TLB[Index].PPN + Offset`
   (d) TLB miss ==¿ Go though page table to get PPN Buffer the result in TLB

In terms of latency, memory access takes hundreds of CPU cycles while TLB access takes only a couple of CPU cycles.

2.12 Multi-set associative TLB

With the TLB with 4 entries only example given in the slides we may end up with two addresses that map to the same TLB index. This will lead to constant caches misses if we access both addresses in an alternating fashion. We will miss the first address, cache it in the TLB, miss the second address, evict the first address in the TLB and cache the second address into the TLB and then again miss the first address and so forth. So we may want to use a Multi-set associative TLB. We split a virtual address as follows

```
 -------------------------------
| Tag | Set Index | Page Offset |
 -------------------------------
 ^   ^ ^         ^ ^           ^
63  16 15       12 11          0
```

where the set index lets us find the set in the TLB we would like to access and the tag corresponds to the mapping from a given VPN to a PPN (note that the tag is smaller than the VPN). A 16 set 4 way associative TLB can hold 64 different PPN's and also avoid the cache missing example given above.

3 Caching

3.1 Principle of Locality

- Temporal locality: If memory location x is referenced, then x will likely be referenced again in the near future.
- Spatial locality f memory location x is referenced, then locations near x will likely be referenced in the near future.
- Idea: Buffer recently accessed data in cache close to CPU.

3.2 Caching Ideas

- Naive caching: cache at the byte granularity level and search the cache for each byte accessed. Problem: High bookkeeping overhead since each cache entry has 8 bytes of address and 1 byte of data.
- Caching at block granularity: Cache one block (cache line) at a time. A typical cache line size is 64 bytes. Advantage:
  - Lower bookkeeping overhead: A cache line has 8 byte of address and 64 byte of data.
  - Exploits spatial locality: Accessing location x causes 64 bytes around x to be cached.

3.3 Direct-mapped cache

Caching at block granularity: each cache line has 64 bytes.

3.4 Multi-way set associative cache

```
 -------------------------------------
| Tag | Set Index | Cahce Line Offset |
 -------------------------------------
 ^   ^ ^         ^ ^                  ^
63  10 9         6 5                  0
```

3.5 Cache line replacement policy

- LFU (least-frequently-used): Replace the line that has been referenced the fewest times over some past time window.
- LRU (least-recently-used): Replace the line that has the furthest access in the past.
- These policies require additional time and hardware.

3.6 Further Optimizing Memory Access

- Current design: address translation → cache access. The two steps are performed sequentially.
- Solution: Use a Virtual Index and a Physical Tag.
  - Use VA to index set, calculate the tag from PA.
  - Cache set lookup is done in parallel with address translation.
  - VA looks like:

    ```
     -------------------------------------
    | --- | Set Index | Cahce Line Offset |
     -------------------------------------
     ^   ^ ^         ^ ^                  ^
    63  10 9         6 5                  0
    ```

    where we use the set index and the cache line offset to find the relevant cache line in the cache.
  - PA looks like

    ```
     -------------------------
    | Tag | Cahce Line Offset |
     -------------------------
     ^   ^ ^                  ^
    63   6 5                  0
    ```

    and we use the tag to complete the address translation.

3.7 Memory Hierarchy

From top to bottom:

- CPU
- L1 Cache (cycles)
- L2 Cache (12 cycles)
- L3 Cache (35 cycles)
- Memory (150 cycles)

3.8 Cache summary

- Caching can speed up memory access
  - L1/L2/L3 cache data
  - TLB caches address translation
- Cache design:
  - Direct mapping
  - Multi-way set associative
  - Virtual index + physical tag parallelize cache access and address translation

3.9 Writing Cache Friendly Code

Take advantage of the locality principles. For example, in C array are ordered in row-major order. For better cache performance it would make more sense to traverse a multi-dimensional array row by row compared to column-by-column. This way the cache will have more cache hits than misses.