# Section 12: Cache, Clock Algorithm, and Demand Paging

CS162
August 5, 2019

## Contents

1. **Vocabulary**

2. **Problems**
   - 2.1 Caching
   - 2.2 Clock Algorithm
   - 2.3 Demand Paging
   - 2.4 Virtual Memory
   - 2.5 Page Allocation
   - 2.6 Address Translation
   - 2.7 Inverted Page Tables
1 Vocabulary

- **Compulsory Miss** The miss that occurs on the first reference to a block. There’s essentially nothing that you can do about this type of miss, but over the course of time, compulsory misses become insignificant compared to all the other memory accesses that occur.

- **Capacity Miss** This miss occurs when the cache can’t contain all the blocks that the program accesses. Usually the solution to capacity misses is to increase the cache size.

- **Conflict Miss** Conflict misses occur when multiple memory locations are mapped to the same cache location. In order to prevent conflict misses, you should either increase the cache size or increase the associativity of the cache.

- **Coherence Miss** Coherence misses are caused by external processors or I/O devices that updates what’s in memory.

- **Working set** The subset of the address space that a process uses as it executes. Generally we can say that as the cache hit rate increases, more of the working set is being added to the cache.

- **Thrashing** Phenomenon that occurs when a computer’s virtual memory subsystem is constantly paging (exchanging data in memory for data on disk). This can lead to significant application slowdown.

- **Inverted Page Table** - The inverted page table scheme uses a page table that contains an entry for each physical frame, not for each logical page. This ensures that the table occupies a fixed fraction of memory. The size is proportional to physical memory, not the virtual address space. The inverted page table is a global structure – there is only one in the entire system. It stores reverse mappings for all processes. Each entry in the inverted table contains a tag containing the task id and the virtual address for each page. These mappings are usually stored in associative memory (remember fully associative caches from 61C?). Associatively addressed memory compares input search data (tag) against a table of stored data, and returns the address of matching data. They can also use actual hash maps.

- **Translation Lookaside Buffer (TLB)** - A translation lookaside buffer (TLB) is a cache that memory management hardware uses to improve virtual address translation speed. It stores virtual address to physical address mappings, so that the MMU can store recently used address mappings instead of having to retrieve them multiple times through page table accesses.
2 Problems

2.1 Caching

An up-and-coming big data startup has just hired you to help design their new memory system for a byte-addressable system. Suppose the virtual and physical memory address space is 32 bits with a 4KB page size.

First, you create 1) a direct mapped cache and 2) a fully associative cache of the same size that replaces the least recently used pages when the cache is full. You run a few tests and realize that the fully associative cache performs much worse than the direct mapped cache. What’s a possible access pattern that could cause this to happen?

Let’s say each cache held X amount of blocks. An access pattern would be to repeatedly iterate over X+1 consecutive blocks, which would cause everything in the fully associated cache to miss.

Instead, your boss tells you to build a 8KB 2-way set associative cache with 64 byte cache blocks. How would you split a given virtual address into its tag, index, and offset numbers?

Since it’s two way set associative, the cache is split into two 4KB banks. The offset will take 6 bits, since $2^6 = 64$. Each bank can store 64 blocks, since $2^{12}/2^6 = 2^6$, so there will be 6 index bits. Then the rest of the bits will be for the tag. It will look like this:

<table>
<thead>
<tr>
<th>20 Bits</th>
<th>6 Bits</th>
<th>6 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Index</td>
<td>Offset</td>
</tr>
</tbody>
</table>

You finish building the cache, and you want to show your boss that there was a significant improvement in average read time. Suppose your system uses a two level page table to translate virtual addresses and your system uses the cache for the translation tables and data. Each memory access takes 50ns, the cache lookup time is 5ns, and your cache hit rate is 90%. What is the average time to read a location from memory?

Since the page table has two levels, there are three reads for each access. For each access, the average access time is: $0.9 \times 5 + 0.1 \times (5 + 50) = 10$ns. Since there are 3 accesses, we multiply this by 3 to get an average read time of 30ns.
2.2 Clock Algorithm

Suppose that we have a 32-bit virtual address split as follows:

<table>
<thead>
<tr>
<th>10 Bits</th>
<th>10 Bits</th>
<th>12 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table ID</td>
<td>Page ID</td>
<td>Offset</td>
</tr>
</tbody>
</table>

Show the format of a PTE complete with bits required to support the clock algorithm.

<table>
<thead>
<tr>
<th>20 Bits</th>
<th>8 Bits</th>
<th>1 Bit</th>
<th>1 Bit</th>
<th>1 Bit</th>
<th>1 Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPN</td>
<td>Other</td>
<td>Dirty</td>
<td>Use</td>
<td>Writable</td>
<td>Valid</td>
</tr>
</tbody>
</table>

For this problem, assume that physical memory can hold at most four pages. What pages remain in memory at the end of the following sequence of page table operations and what are the use bits set to for each of these pages:
- Page A is accessed
- Page B is accessed
- Page C is accessed
- Page A is accessed
- Page C is accessed
- Page D is accessed
- Page B is accessed
- Page D is accessed
- Page A is accessed
- Page E is accessed
- Page F is accessed

E: 1, F: 1, C: 0, D: 0
2.3 Demand Paging

Your boss has been so impressed with your work designing the caching that he has asked for your advice on designing a TLB to use for this system. Suppose you know that there will only be 4 processes running at the same time, each with a Resident Set Size (RSS) of 512MB and a working set size of 256KB. Assuming the same system as the previous problem (32 bit virtual and physical address space, 4KB page size), what is the minimum amount of TLB entries that your system would need to support to be able to map/cache the working set size for one process? What happens if you have more entries? What about less?

A process has a working set size of 256KB which means that the working set fits in 64 pages. This means our TLB should have 64 entries. If you have more entries, then performance will increase since the process often has changing working sets, and it should be able to store more in the TLB. If it has less, then it can’t easily translate the addresses in the working set and performance will suffer.

Suppose you run some benchmarks on the system and you see that the system is utilizing over 99% of its paging disk IO capacity, but only 10% of its CPU. What is a combination of the of disk space and memory size that can cause this to occur? Assume you have TLB entries equal to the answer from the previous part.

The CPU can’t run very often without having to wait for the disk, so it’s very likely that the system is thrashing. There isn’t enough memory for the benchmark to run without the system page faulting and having to page in new pages. Since there will be 4 processes that have a RSS of 512MB each, swapping will occur as long as the physical memory size is under 2GB. This happens regardless of the number of TLB entries and disk size. If the physical memory size is lower than the aggregate working set sizes, thrashing is likely to occur.

Out of increasing the size of the TLB, adding more disk space, and adding more memory, which one would lead to the largest performance increase and why?

We should add more memory so that we won’t need to page in new pages as often.
### 2.4 Virtual Memory

`vmstat` is a Linux performance debugging tool that provides information about virtual memory on your system. When you run it, the output looks like this:

```
$ vmstat 1
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
       r  b   swpd free  buff   cache  si  so  bi  bo  in  cs us  sy  id  wa  st
1 0  0 174184 1007372  96316  49  642 3095 678 123 128  0  1  99  0  0
0 0  0 174240 1007372  96316  0  0  0  0  48  88  0  0 100  0  0
0 0  0 174240 1007372  96316  0  0  0  0  33  75  0  0 100  0  0
0 0  0 174240 1007372  96316  0  0  0  0  32  73  0  0 100  0  0
```

The `1` means “recompute the stats every 1 second and print them out”. The first line contains the average values since boot time, while the second line contains the averages of the last second (current averages). Here’s a reference for what each one of the columns means.

**Procs**
- **r**: The number of runnable processes (running or waiting for run time).
- **b**: The number of processes in uninterruptible sleep.

**Memory**
- **swpd**: the amount of virtual memory used.
- **free**: the amount of idle memory.
- **buff**: the amount of memory used as buffers.
- **cache**: the amount of memory used as cache.
- **inact**: the amount of inactive memory. (**-a** option)
- **active**: the amount of active memory. (**-a** option)

**Swap**
- **si**: Amount of memory swapped in from disk (/s).
- **so**: Amount of memory swapped to disk (/s).

**IO**
- **bi**: Blocks received from a block device (blocks/s).
- **bo**: Blocks sent to a block device (blocks/s).

**System**
- **in**: The number of interrupts per second, including the clock.
- **cs**: The number of context switches per second.

**CPU**
- These are percentages of total CPU time.
  - **us**: Time spent running non-kernel code. (user time, including nice time)
  - **sy**: Time spent running kernel code. (system time)
  - **id**: Time spent idle. Prior to Linux 2.5.41, this includes IO-wait time.
  - **wa**: Time spent waiting for IO. Prior to Linux 2.5.41, included in idle.
  - **st**: Time stolen from a virtual machine. Prior to Linux 2.6.11, unknown.
Take a look at these 3 programs (A, B, C).

```c
char *buffer[4 * (1L << 20)];
int A(int in) {
    // "in" is a file descriptor for a file on disk
    while (read(in, buffer, sizeof(buffer)) > 0);
}

int B() {
    size_t size = 5 * (1L << 30);
    int *x = malloc(size);
    memset(x, 0xCC, size);
}

sem_t sema;
pthread_t thread;
void *foo() { while (1) sem_wait(&sema); }

int C() {
    pthread_create(&thread, NULL, foo, NULL);
    while (1) sem_post(&sema);
}
```

I ran these 3 programs one at a time, but in a random order. What order did I run them in? Can you tell where (in the vmstat output) one program stopped and another started? Explain.

I ran C (rows 2-6), B (rows 8-13), then A (rows 15-20). Program C has a lot of context switches (cs), but no swap or IO activity. Both program A and B have a lot of disk I/O, but program A is read-only IO (bi) and program B is the only one that should be swapping to disk (so).
If you have extra available physical memory, Linux will use it to cache files on disk for performance benefits. This disk cache may also include parts of the swapfile. Why would caching the swapfile be better than paging-in those pages immediately?

If the system’s memory needs grow, the disk cache must shrink. If we avoid paging-in those cached parts of the swapfile, we can immediately evict the swapfile from memory without needing to modify any page tables.

If I remove the line “memset(x, 0xCC, size);” from program B, I notice that the vmstat output does not have a spike in swap (si and so) nor in io (bi and bo). My system doesn’t have enough physical memory for a 5GB array. Yet, the array is not swapped out to disk. Where does the array go? Why did the memset make a difference?

The memory returned by malloc is sparsely allocated. It won’t actually be allocated until I begin using it. The memset will force the kernel to materialize those pages, because I write to them.

Program B has a 5GB array, but the whole thing just contains 0xCCCCCCCC. Based on this observation, can you think of a way to reduce program B’s memory footprint without changing any of program B’s code? (What can the kernel do to save memory?)

The kernel could compress pages in memory. It’s usually faster to uncompress memory than it is to page-in from disk. If the entire array is the same byte repeated over and over, then the kernel could achieve very good compression ratios on the array. The kernel must mark infrequently-used pages as invalid, so that it has a chance to uncompress those pages inside the page fault exception handler.
2.5 Page Allocation

Suppose that you have a system with 8-bit virtual memory addresses, 8 pages of virtual memory, and 4 pages of physical memory.

How large is each page? Assume memory is byte addressed.

32 bytes

Suppose that a program has the following memory allocation and page table.

<table>
<thead>
<tr>
<th>Memory Segment</th>
<th>Virtual Page Number</th>
<th>Physical Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A</td>
<td>000</td>
<td>NULL</td>
</tr>
<tr>
<td>Code Segment</td>
<td>001</td>
<td>10</td>
</tr>
<tr>
<td>Heap</td>
<td>010</td>
<td>11</td>
</tr>
<tr>
<td>N/A</td>
<td>011</td>
<td>NULL</td>
</tr>
<tr>
<td>N/A</td>
<td>100</td>
<td>NULL</td>
</tr>
<tr>
<td>N/A</td>
<td>101</td>
<td>NULL</td>
</tr>
<tr>
<td>N/A</td>
<td>110</td>
<td>NULL</td>
</tr>
<tr>
<td>Stack</td>
<td>111</td>
<td>01</td>
</tr>
</tbody>
</table>

What will the page table look like if the program runs the following function? Page out the least recently used page of memory if a page needs to be allocated when physical memory is full. Assume that the stack will never exceed one page of memory.

What happens when the system runs out of physical memory? What if the program tries to access an address that isn’t in physical memory? Describe what happens in the user program, the operating system, and the hardware in these situations.

```c
#define PAGE_SIZE 1024; // replace with actual page size

void helper(void) {
    char *args[5];
    int i;
    for (i = 0; i < 5; i++) {
        // Assume malloc allocates an entire page every time
        // Assume malloc allocates an entire page every time
        args[i] = (char*) malloc(PAGE_SIZE);
    }
    printf("%s", args[0]);
}
```

<table>
<thead>
<tr>
<th>Memory Segment</th>
<th>Virtual Page Number</th>
<th>Physical Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>000</td>
<td>00</td>
</tr>
<tr>
<td>Code Segment</td>
<td>001</td>
<td>10</td>
</tr>
<tr>
<td>Heap</td>
<td>010</td>
<td>11</td>
</tr>
<tr>
<td>N/A</td>
<td>011</td>
<td>NULL</td>
</tr>
<tr>
<td>N/A</td>
<td>100</td>
<td>NULL</td>
</tr>
<tr>
<td>N/A</td>
<td>101</td>
<td>NULL</td>
</tr>
<tr>
<td>N/A</td>
<td>110</td>
<td>NULL</td>
</tr>
<tr>
<td>Stack</td>
<td>111</td>
<td>01</td>
</tr>
<tr>
<td>Memory Segment</td>
<td>Virtual Page Number</td>
<td>Physical Page Number</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>Heap</td>
<td>000</td>
<td>00</td>
</tr>
<tr>
<td>Code Segment</td>
<td>001</td>
<td>10</td>
</tr>
<tr>
<td>Heap</td>
<td>010</td>
<td>PAGEOUT</td>
</tr>
<tr>
<td>Heap</td>
<td>011</td>
<td>11</td>
</tr>
<tr>
<td>N/A</td>
<td>100</td>
<td>NULL</td>
</tr>
<tr>
<td>N/A</td>
<td>101</td>
<td>NULL</td>
</tr>
<tr>
<td>N/A</td>
<td>110</td>
<td>NULL</td>
</tr>
<tr>
<td>Stack</td>
<td>111</td>
<td>01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Segment</th>
<th>Virtual Page Number</th>
<th>Physical Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>000</td>
<td>PAGEOUT</td>
</tr>
<tr>
<td>Code Segment</td>
<td>001</td>
<td>10</td>
</tr>
<tr>
<td>Heap</td>
<td>010</td>
<td>PAGEOUT</td>
</tr>
<tr>
<td>Heap</td>
<td>011</td>
<td>11</td>
</tr>
<tr>
<td>Heap</td>
<td>100</td>
<td>00</td>
</tr>
<tr>
<td>N/A</td>
<td>101</td>
<td>NULL</td>
</tr>
<tr>
<td>N/A</td>
<td>110</td>
<td>NULL</td>
</tr>
<tr>
<td>Stack</td>
<td>111</td>
<td>01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Segment</th>
<th>Virtual Page Number</th>
<th>Physical Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>000</td>
<td>PAGEOUT</td>
</tr>
<tr>
<td>Code Segment</td>
<td>001</td>
<td>10</td>
</tr>
<tr>
<td>Heap</td>
<td>010</td>
<td>PAGEOUT</td>
</tr>
<tr>
<td>Heap</td>
<td>011</td>
<td>PAGEOUT</td>
</tr>
<tr>
<td>Heap</td>
<td>100</td>
<td>00</td>
</tr>
<tr>
<td>Heap</td>
<td>101</td>
<td>11</td>
</tr>
<tr>
<td>N/A</td>
<td>110</td>
<td>NULL</td>
</tr>
<tr>
<td>Stack</td>
<td>111</td>
<td>01</td>
</tr>
</tbody>
</table>
2.6 Address Translation

Consider a machine with a physical memory of 8 GB, a page size of 8 KB, and a page table entry size of 4 bytes. How many levels of page tables would be required to map a 46-bit virtual address space if every page table fits into a single page?

Since each PTE is 4 bytes and each page contains 8 KB, then a one-page page table would point to 2048 or \(2^{11}\) pages, addressing a total of \(2^{11} \times 2^{13} = 2^{24}\) bytes.

- Depth 1 = \(2^{24}\) bytes
- Depth 2 = \(2^{35}\) bytes
- Depth 3 = \(2^{46}\) bytes

So in total, 3 levels of page tables are required.

List the fields of a Page Table Entry (PTE) in your scheme.

Each PTE will have a pointer to the proper page, PPN, plus several bits – read, write, execute, and valid. This information can all fit into 4 bytes, since if physical memory is \(2^{33}\) bytes, then 20 bits will be needed to point to the proper page, leaving ample space (12 bits) for the information bits.

Without a cache or TLB, how many memory operations are required to read or write a single 32-bit word?

Without extra hardware, performing a memory operation takes 4 actual memory operations: 3 page table lookups in addition to the actual memory operation.

With a TLB, how many memory operations can this be reduced to? Best-case scenario? Worst-case scenario?

- Best-case scenario: 2 memory lookups. once in TLB, once for actual memory operation.
- Worst-case scenario: 5 memory lookups. once in TLB + 3 page table lookups in addition to the actual memory operation.

The pagemap is moved to main memory and accessed via a TLB. Each main memory access takes 50 ns and each TLB access takes 10 ns. Each virtual memory access involves:

- mapping VPN to PPN using TLB (10 ns)
- if TLB miss: mapping VPN to PPN using page map in main memory (50 ns)
- accessing main memory at appropriate physical address (50 ns)

Assuming no page faults (i.e. all virtual memory is resident) what TLB hit rate is required for an average virtual memory access time of 61 ns.

\[(10+50)*x+(1-x)*(50+10+50) = 61\]

solve for x gives \(x = .98 = 98\%\) hit rate

Assuming a TLB hit rate of .50, how does the average virtual memory access time of this scenario compare to no TLB?

With a TLB with a hit rate of 0.5:

\[x = 0.5\]

\[\text{avg.

Without a TLB:
time = 50 + 50
time = 100
### 2.7 Inverted Page Tables

Why IPTs? Consider the following case:
- 64-bit virtual address space
- 4 KB page size
- 512 MB physical memory

How much space (memory) needed for a single level page table? Hint: how many entries are there? 1 per virtual page. What is the size of a page table entry? access control bits + physical page #.

One entry per virtual page

- $2^{64}$ addressable bytes / $2^{12}$ bytes per page = $2^{52}$ page table entries

Page table entry size

- 512 MB physical memory = $2^{29}$ bytes
- $2^{29}$ bytes of memory / $2^{12}$ bytes per page = $2^{17}$ physical pages
- 17 bits needed for physical page number
- Page table entry = ~4 bytes
- 17 bit physical page number = ~3 bytes
- Access control bits = ~1 byte

Page table size = page table entry size * # total entries

- $2^{52}$ page table entries * $2^{2}$ bytes = $2^{54}$ bytes (16 petabytes)

i.e. A WHOLE LOT OF MEMORY

How about multi level page tables? Do they serve us any better here?

What is the number of levels needed to ensure that any page table requires only a single page (4 KB)?

- Assume page table entry is 4 bytes
- 4 KB page / 4 bytes per page table entry = 1024 entries
- 10 bits of address space needed
- ceiling(52/10) = 6 levels needed

7 memory accesses to do something? SLOW!!!

Linear Inverted Page Table

What is the size of of the hashtable? What is the runtime of finding a particular entry?

Assume the following:
- 16 bits for process ID
- 52 bit virtual page number (same as calculated above)
- 12 bits of access information

- add up all bits = 80 bits = 10 bytes
- 10 bytes * # of physical pages = 10 * $2^{17}$ = $2^{3}$ * $2^{17}$ = 1 MB

Iterate through all entries.
For each entry in the inverted page table, compare process ID and virtual page number in entry to the requested process ID and virtual page number. Extremely slow. Must iterate through $2^{17}$ entries of the hash table worst-case scenario.

Hashed Inverted Page Table
What is the size of the hashtable? What is the runtime of finding a particular entry?
Assume the following:
- 16 bits for process ID
- 52 bit virtual page number (same as calculated above)
- 12 bits of access information

\{ add up all bits = 80 bits = 10 bytes \\
- 10 bytes \times \# of physical pages = 10 \times 2^{17} = 2^3 \times 2^{17} = 1 \text{ MB} \}

Linear inverted page tables require too many memory accesses.
- Keep another level before actual inverted page table (hash anchor table)
\{ Contains a mapping of process ID and virtual page number to page table entries \}
- Use separate chaining for collisions
- Lookup in hash anchor table for page table entry
\{ Compare process ID and virtual page number \\
- if match, then found \\
- if not match, check the next pointer for another page table entry and check again \}

So, with a good hashing scheme and a hashmap proportional to the size of physical memory, $O(1)$ time. Very efficient!