Summary of chapter 9:

**What is a cache?**

A cache is often a small memory that stores data. The purpose of the cache is that future requests of data can go faster by avoiding going all the way down to the memory. The cache is more faster and will return the data much faster.

It is also essential to mention that cache is a copy of data from the memory. We don’t transfer the original data from memory, but holds a copy of that data.

**What are the different types of hardware caches?**

We have different types of caches;

* TLB (translation lookaside buffer) which stores references to page frames that programs tend to reference a lot.
* Virtually addressed cache where each entry in the cache stores the memory value associated with the virtual address.
* Physically addressed cache where each entry in the cache stores the value associated with a physical address.

**How do we know if a cache is worth using?**

As mentioned, we want it to be faster fetching data from the cache rather than going to the memory and fetch the data from there. We want the cache hit rate to be as high as possible.

At the same time, the hit rate must be high enough to make it worth the effort. One source of predictability is temporal locality. Temporal locality says that it is likely that a program tends to reference the same memory locations multiple times.

Another one is spatial locality. That says that a program tends to reference nearby locations.

**What is a program’s working set?**

A working set is the amount of memory that a process requires in a given time interval. As long as the working set can fit in the cache, most references will be a cache hit, and the performance of the application will be good.

So a working set is a set of memory locations that need to be cached for reasonable cache hit rate.

The set of memory locations that a program has referenced in the recent past.

**What is thrashing?**

A program trashes if the cache is too small to hold it’s working set. In other words the system has too small cache. Therefore, it is most likely we will have several cache misses as the cache is too small to hold the working set necessary to maintain a high cache hit rate.

**How is concept of cache read?**

The CPU will do a memory access, then look in the cache. If it is a hit, it will store the value in the cache. If it is a miss, the CPU will fetch the address. This is for **virtual** **addresses**.

**How is concept of cache write?**

???????????????

**What is demand paging?**

Demand paging is a type of swapping done in virtual memory. In demand paging, the data is not copied from disk to RAM before it is needed or being demanded by some program. In other words it is copied into physical memory only

With swapping we say that we swap the data from the secondary storage to the main memory.

Demand paging is with a page table implementation. The page table contains a pointer to the page frame in physical memory as well as the access permission for that specific page/frame. However, the access permission part of the page table is a bitwise operator to mark if that page is valid or invalid. If is valid it resides in main memory (RAM?). If it is invalid it does not lie within the main memory, but in the secondary storage.

**So what happens when the CPU tries to access a virtual page/physical frame that is marked as invalid?**

As it is invalid, it is not stored in the main memory. We will first have an TLB miss, then walk through the page table until we hit the page. This is marked as invalid, and we will then trap to the kernel. Convert that virtual address to file + offset. Then we will allocate the page frame in physical memory. Initiate disk block read into page frame. Then the direct memory access. When that is completed we will mark the page as valid as it is now in main memory.

**MEMORY MAPPED FILES AND DEMAND PAGING NEEDS TO GET MORE CLEAR!!!!**

**What are the different page replacement policies?**

First and foremost, the goal of the cache replacement policy is to reduce cache misses.

Do we replace an entry everytime we get an cache miss?

So we have different ways of thinking when it comes to replacing those entries in cache.

**FIFO** stands for first in first out. The first of the entries in the cache, will be the first to be replaced. So it will be the one being in the cache for the longest time.

**MIN**, or **optimal** is the absolutely best way to do it. Unfortunately, this policy require us to look into the future. It will replace the cache entry that will not be used for the longest time. This is of course not possible, but gives us something to compare to.

**LRU** stands for least recently used and will replace the cache entry that has not been used for the longest time. It has therefore been there for a time without being used. This method is approximate to MIN.

**LFU** stands for least frequently used. This will replace the cache entry that has been used the least often.

**Clock algorithm,** is also a way to do it. This will periodically sweep through all physical memory pages like a clock. If a page has not been used, reclaim it (?).

If the page is used, mark it as unused. ???

**What are the different cache types?**

**Fully associative cache:**

The address can be stored anywhere in the table (cache). In that case, on a lookup (checking the cache) the system must check every entry in the table. Therefore it will take long time because it needs to go through the whole.

On the other side, this provides huge flexibility as the addresses can be stored anywhere in the cache. There are no restrictions on where to store them.

**Direct mapped:**

On direct mapped each address can only be stored in one location in the table. This is a more efficient way as it takes less time each cache lookup. We lookup by hash the address to its entry. Therefore, it is a cache hit if the address match that entry, and a miss otherwise.

There is however decreased flexibility as there are more restrictions.

***We don’t need a replace algorithm here. Why is that????!!***

**Set associative:**

With set associative we split up the hash address into two. We replicate the direct mapped table and lookup in each replica in parallel. A k set assosiative cache holds k replicas; a particular block can be in any of the k replicas. There is a cache hit if the address matches any of the replicas.