8.1 Lecture Summary

- Virtual Memory vs. Physical Memory
- mmaped I/O
- TLB / Thrashing
- Paging / Thrashing
- Shared Memory

8.2 Virtual Memory

Memory are divided into page in which the physical address is mapped to a virtual address. Processes use virtual addresses which start at 0, and are allocated lazily onto pages.

8.2.1 Memory Management Unit (MMU)

The MMU is in charge of translating virtual addresses to their physical address. It maintains a page table (big hash table) to map the translations. This is unfortunately too slow when it comes to looking up the addresses, so we want the table to also be in the cache. The cache is about 100 times faster than RAM.

8.2.2 Translation Lookaside Buffer (TLB)

The alternative is a TLB which is a fully associative cache (expensive) where the size is limited. It caches the records used for page translations, the page number and the frame. The powers of 2 is used to simplify virtual to physical address translations. The virtual space size is $2^m$, and page size $2^n$. The high order $m - n$ of the address is used to select a page, and the low order selects the offset.

Chances of a miss in the TLB is high due to its small size (8 to 2048 entries). The assumption made is once a page is touched, it will be in use for awhile. If the TLB was larger, it would improve the hit ratio and decrease average memory cost.

Every process has its own set of mapping (page hash table), meaning each process has its own associated table or TLB. When changing processes, the TLB of the old process is to be deleted, and a new TLB is created for the new process in a new table. This process is known as TLB shutdown. Note that the the old TLB are not stored in cache.

**TLB thrashing** occurs when it visits pages all over the place in and out between the cache and disk memory that it slows down the entire process. The working set exceeds the size of TLB.
8.2.3 Benefits of Virtual Memory

- **Simpler**: Everyone gets the illusion of that it is the whole address space. There’s no need to worry about who else may be accessing that same memory, it can be treated as if it’s all yours.

- **Segments**: One person only manages one small chunk. If a bigger chunk is needed, must fetch it manually.

- **Isolation**: Every process is protected from every other. No read or write is allowed to or from other processes. There is **Stability** if it tries to do this, a segmentation violation is given so that it doesn’t overwrite something else. There are 2 bits stored in the TLB that distinguishes whether the memory is read and writable (1 bit each), and sometimes an extra bit for exe.

- **Optimization**: It reduces space requirements.

8.3 Sharing Memory

Paging allows the sharing of memory across processes, significantly reducing memory requirements. As long as all processes accessing the same single page in memory is not altering or writing into the memory, then it can be viewed (read-only) by all processes. Hence, code is always read-only while data is copy-on-write. However, sharing requires the hardware to be fast.

Static libraries in C are not shared, everyone has their own copy if more than one process uses it.

8.4 Life of A Page

Allocate memory, use it, evict it (some memory written to disk due to lack of memory), and then reuse it.

8.4.1 Allocate Memory

Allocate memory by using `mmap` to obtain memory directly from the OS. `mmapped I/O` has no physical frame allocated at all until it starts to touch the pages. The address space is `reserved` since memory is not committed to physical memory yet meaning it is still free. Once the page is touched, it becomes `committed` address space now that the physical memory is in use and is not free. Committing space is done *lazily*.

8.4.2 Using Memory

When memory is used, the page tables are updates and when contents are written, it is a dirty page.

- Linux, LRU page evicted to disk
- Windows, LRU by program evicted to a pool

If touching a page, may not use extra memory (read-shared), writing-used.
8.4.3 Evicting Memory

*Paged out* is forcing our page to be evicted. It saves whatever is in the **swap space (disk)** and then gives up the physical memory. Read/Write from this virtual memory is no longer possible. When another page is touched, it is swapped in. If you not desire certain memory to be swapped, you can pin the memory. **Pin memory** locks the memory and becomes self manageable. (A random tangent I wrote down in this section: in-kernel web servers avoids context switch.) Otherwise, it has replacement policies.

8.5 Paging

8.5.1 Cost of Paging

Worst-case analysis is useless. Easy to construct an adversary where every page requested is a **page fault** such as having A,B,C,D,E,F,G,H,I,J,A... being read by 5 bits. There are infinite faults and it doesn’t matter which is evicted.

8.5.2 Competitive Analysis

Compare to optimal replacement policy: k-ratio where k is the size of queue - LRU and FIFO. It’s no worse than 5 times future of page faults. Can incur k times more misses than OPT, no more than than 20,000 times worse than the OPT.

8.5.3 MIN aka OPT (Optimal Page Replacement)

Evict page accessed furthest in the future. This is similar to the idea of oracles. Trace and simulate (imperative studies): it runs through and simulates (make predictions) of the future from the known.