COMP124

Computer Systems

19 | Paging | Working Sets | Caches

Dr Stuart Thomason

Paging Memory Model

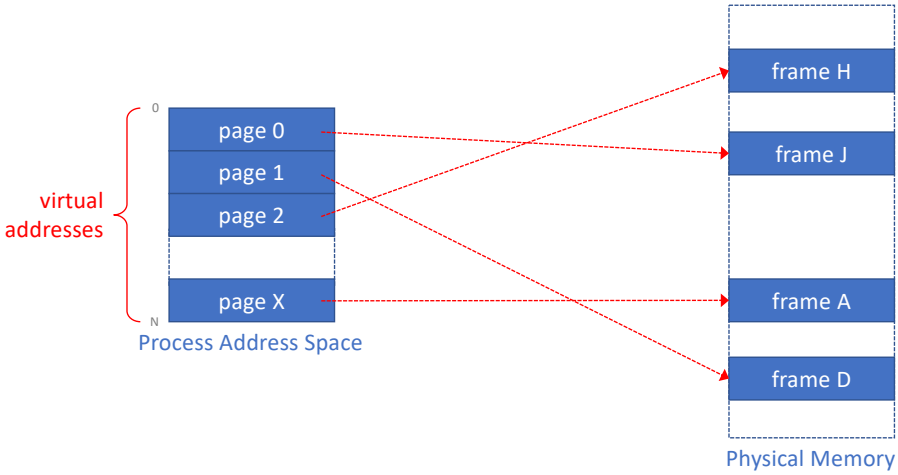
- Paging splits physical memory into equal-sized blocks
  - Process address space spread across as many pages as necessary
  - Very similar to how files are stored across blocks on a disk

- Process memory addresses will appear to be one contiguous range
  - Logical addresses numbered from 0 to N (process limit)
  - In reality they will be spread over multiple pages
  - The pages could be anywhere in memory (do not need to be consecutive)
  - The physical memory used by a page is called the page frame

- Each process has a page table that maps logical pages onto physical memory frames
  - Needs to be stored in memory as part of the process image
  - Address of table stored in page table base register within the CPU

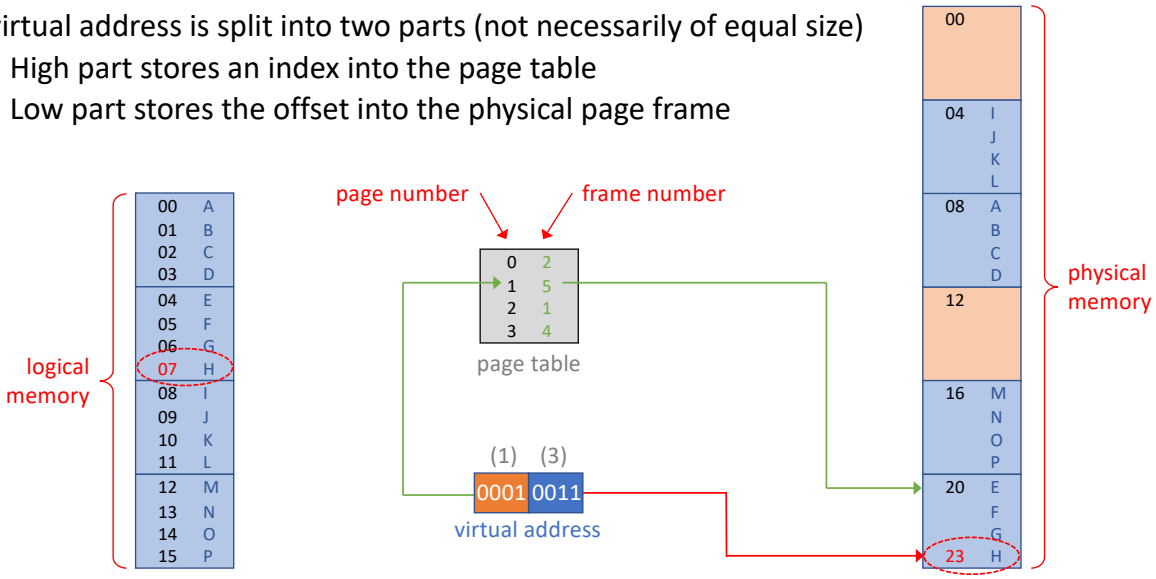
Page Mapping

- Frames can be anywhere in the real physical memory
- But the virtual pages look like a contiguous address space



Page Table Lookups

- A virtual address is split into two parts (not necessarily of equal size)
  - High part stores an index into the page table
  - Low part stores the offset into the physical page frame



Segmentation and Paging

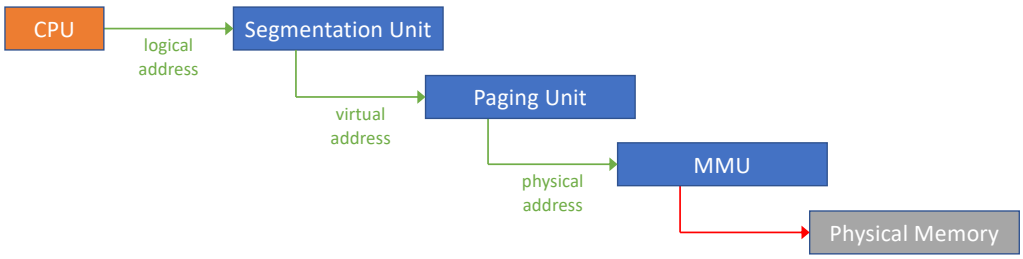
- Segmentation…
  - Logical division of address space
  - Segments can be different sizes
  - Segments match the structure of the program

- Paging…
  - Physical division of address space
  - Each page is the same size
  - Pages have no relation to program structure

- Advantages of paging…
  - Memory allocation is much easier with fixed size units (pages)
  - Fragmentation is eliminated (but with some wasted space in final page of a process)

Segmentation with Paging

- The Intel x86 MMU supports segmentation with paging
  - Process memory is split into logical segments
  - Segments are stored inside pages
  - Concepts from previous lecture still apply (but within a paged memory layout)



- CPU is doing this every time an instruction needs to access memory to get or store data

Page Swapping

- The OS can allocate more memory than is physically installed in the system
  - Only pages that are used by the current process need to be in main memory
  - Other pages can be stored on disk (in the swap space)
  - Allows the system to have as many processes as it likes in the round robin queue
  - But the OS doesn't necessarily know which pages will be needed at any time
- If a process tries to access (eg. jump, read, write) a page that's not in main memory…
  - A page fault occurs
  - This is not an error (although it sounds like it)
- Every page fault generates an interrupt
  - Interrupt handler catches the page fault
  - Correct page is loaded (or swapped) from disk into main memory
  - Known as demand paging (pages are brought in when needed)
  - Some systems use anticipatory paging (bringing in more pages than needed)

Page Replacement Problem

- When we bring a page from disk into memory…
  - Might need to make room for it by swapping out an unused page
  - How do we choose which page to swap out to disk?

- Several policies exist
  - Least recently used (LRU) – the page that has been unused for the longest period
  - First in first out (FIFO) – the page that has been in memory for the longest period
  - Least frequently used (LFU) – the page with the fewest references in recent period
  - Second chance – uses FIFO but keeps pages that were recently used

- The memory manager might get it wrong
  - Can only make a sensible guess based on previous process history and page usage
  - Just leads to more page faults and disk "thrashing" as pages are swapped

Principle of Locality

- The optimal policy is to swap out the page that won't be needed for the longest time
  - Can only ever be based on a best guess
  - Cannot predict the future
  - But can use the principle of locality as a guide
    - "Over any short period of time, a process's memory references tend to be spatially localised"
  - In other words, memory addresses that a process refers to in the next time period are likely to be close to the addresses used in the current period
  - Cannot always be guaranteed, so page faults are a normal part of how an operating system works (ie. they are not error conditions)

Working Set

- Following the principle of locality…
  - Instructions don't usually jump all over the place
  - Code within a loop spends time in the same area of memory
  - Accessing an array or other data structure will be focused on the same memory area
- Generally, execution stays within 20% of a process's pages for 80% of the time (to simplify things, on this module the time period is not specified in units)
- The working set of a process is defined as the set of pages W(T, s) used in the time period from T to T + s
- For example, W(12, 4) is the working set of pages used by a process in the time period from 12 to 16 (inclusive)

Predicting the Working Set

- Memory manager knows which pages have been used in which time periods

Page	z	a	b	a	a	a	z	b	b	a
Time	0	1	2	3	4	5	6	7	8	9

- What is the working set W(3, 3)? – **az**

- By the principle of locality, the working set for the next time period is likely to be similar to the one for the current period
- For each process, the memory manager…
  - Tries to ensure its working set is in memory
  - Estimates its next working set by looking at the set used in the preceding period

- What is the predicted working set W(10, 2)? – **ab**

Working Set Accuracy

- Accuracy of the working set depends on its size
  - Too small – won't cover the entire current locality for a process
  - Too big – will cover several localities

- Working set of a process will change as execution moves from one locality to another
  - References to different data structures
  - Subroutine calls to different parts of the code

- Number of page faults will increase during this transition between localities (and this is an expected part of how memory management works)

Page Size

- Large pages require a smaller page table but increase the amount of wasted memory
  - The final page for each process won't be entirely full
  - On average we lose 50% of a page per process due to internal fragmentation

- Smaller pages make better use of memory and swap space
  - Less fragmentation and wastage
  - Bring in only the code and data needed for a working set

- But small pages increase the chance of page faults
  - Code and data are more likely to be in another page
  - More pages to be swapped in and out

- Windows and Linux both use 4 KB pages by default

Cache Memory

- Storage hierarchy decreases in speed (and cost) as we move away from the CPU
  - The fastest parts of the system are inside the CPU itself (registers)
  - The slowest parts are the mechanical disk drives (HDDs)
  - Main memory is slow compared to registers, but way faster than any disk

- Efficiency (speed) can be improved by keeping things as close to the CPU as possible

- The cache is a small, fast memory area that sits between the CPU and main memory
  - Stored inside the CPU itself
  - Contains copies of items in main memory

- Device controllers might also have their own cache memory to speed up data transfer (but these operate entirely separately from the main CPU cache)

Cache Hits and Misses

- When the CPU requests data from memory, it first checks the cache
- If the data is in the cache…
  - This is a cache hit
  - CPU can immediately read the data from the cached copy (very fast)
- If the data is not in the cache…
  - This is a cache miss
  - MMU will copy a data block from main memory into the cache (with a time penalty)
  - Block will include memory locations around the requested address
  - CPU can then read the data from this cached version
- By the principle of locality…
  - It's likely that future data items were brought into the cache as part of the block
  - Chances of cache hits are improved

Cache Levels and Latency

- Modern processors are incredibly fast
- Memory access is incredibly slow by comparison
- CPU can execute hundreds of instructions in the time it takes to fetch a data block from main memory into the cache (latency)
- Larger caches have better hit rates but longer latency

- Modern CPUs have multiple levels of cache
  - Small and fast cache (L1) closest to the CPU registers
  - Backed up by bigger, slower caches (L2 and L3)
- In multi-core processors…
  - Each core has its own L1 cache
  - Each pair of cores shares a slightly bigger L2 cache
  - All cores share a larger L3 cache

Memory Hierarchy