Virtual Memory

Physical memory is your actual RAM. Virtual memory is an abstraction of physical memory, which maps some collection of physical memory to a uniform virtual address range (typically 0x0 -> 0xffffffff for 32bit machines).

A process uses the virtual addresses (unknowingly) which isolates it from other processes, the kernel, and from the actual physical RAM (which may be disjoint) it uses. This abstraction also lets us have more processes running than can fit into RAM.

P physical addresses -> 2^P bytes physical memory
V virtual addresses -> 2^V bytes virtual memory

Dynamic Relocation
-the kernel is responsible for managing MMU registers on address space switches (context switch from thread in one process to thread in a different process)
-the OS must allocate/deallocate variable-sized chunks of physical memory
-potential for *fragmentation* of physical memory

Address translation: translate a virtual address v to a physical address p
if v >= limit then generate exception
else p = v + physical offset(relocation register)
Translation is done in hardware by the memory management unit (MMU)

Segmentation
provide a separate mapping for each segment of the virtual memory that application actually uses
with K bits for the segment ID -> can have up to 2^K segments and 2^(V-K) bytes per segment

translating segmented virtual address
Ri - relocation offset; Li - limit offset for the ith segment
split p into segment number(s) and address within segment(a)
if a >= Ls then generate exception
else p = a + Ri

#bits required for segment number = log(#segments)
segment offset bits = virtual - log(#segments)
segment offset = virtual address - segment size * cur segmentNumber

Single-level paging:
physical memory is divided into fixed-size chunks called frames/physical pages
Virtual memories are divided into fixed-size chunks called pages
Each page maps to a different frame. Any page can map to any frame.

The PAGE TABLE contains the mappings from pages to physical PAGE FRAMES where the size of the page frame is the same as the page size.

Translation lookaside buffer (TLB) - a small, fast, dedicated cache of address translations, in the MMU – Each TLB entry stores a (page# → frame#) mapping

#frames = physical memory size / frame size

#pages = virtual memory size / page size
#page table entries(PTEs) = virtual memory size / page size = #pages
page table size = #PTE * PTE size = virtual memory size / Page size * PTE size
#BITS for PAGE NUM = log(#PAGES)
#BITS for FRAME NUM = log(#FRAMES) = bits for each PTE
#BITS for Page Offset = log(PAGESIZE)
#BITS for Frame Offset = log(FRAMESIZE)

Multi-level paging
two-level paging:
each virtual address has three parts:
level one page number - index the directory
level two page number - index a page table
offset within the page

Properties:
– Can map large virtual memories by adding more levels.
– Individual page tables/directories can remain small.
– Can avoid allocating page tables and directories that are not needed for programs that use a small amount of virtual memory.
– TLB misses become more expensive as the number of levels goes up, since more directories must be accessed to find the correct PTE.

Downside:
Becomes a little bit more expensive to find the address
e.g.: 4-level -> go to RAM 4 times

physAddr = Frame# * FrameSize + page offset
#PTE per page = page size / PTE size
#PageTables = #pages / PTEs per page
#bits for Page Number = log(#PTE per page)
#levels = ceiling[(#VBITS - #PageOffsetBits) / PageNumberBits]

For code & data segment, calculate "top" (i.e. end of segment)
"top" = vbase + npages * pgsize
For stack,
"top" = USERSTACK = 0x8000 0000
calculate "base" = TOP - STACKPGS * PGSIZE
Given vaddr v:
PhysAddr = (v - vbase) + pbase

Secondary storage
Goal:
-Allow virtual address spaces that are larger than the physical address space
-Allow greater multiprogramming levels by using less of the primary memory for each process
Method:
-Allow pages from virtual memories to be stored in secondary storage. i.e.: on disks or SSDs
-swap pages(or segments) between secondary storage and primary memory so that they are in primary memory when they are needed
Disadvantage: slow

Page Fault - kernel will do:
1.swap the page into memory from secondary storage, evicting another page from memory if necessary
2.update the PTE(set the present bit)
3.return from the exception so that the application can retry the virtual memory access that caused the page fault

Performance with **swapping** - try to ensure the page faults are rare
-limit the number of processes, so that there is enough physical memory per process
-try to be smart about which pages are kept in physical memory, and which are evicted
-hide latencies, e.g.: by prefetching pages before a process needs them

FIFO - replace the page that has been in memory the longest
Optimal Page Replacement - replace the page that will not be referenced for the longest time. (not possible in realty, can be used to determine the lower bound of #page faults)

Least Recently Used Page Replacement - replace the least recently used page
MMU can help by tracking page accesses in hardware
Add a use bit to each PTE, which
-is set by the MMU each time the page is used, i.e., each time the MMU translates a virtual address on that page
-can be read and cleared by the kernel

The Clock Replacement Algorithm - is identical to FIFO, except a page is skipped if its use bit is set

```
While use bit of victim is set
        clear use bit of victim
        victim = (victim + 1) % num_frames
choose victim for replacement
victim = (victim + 1) % num_frames
```

I/O

Disk speed: S rpm = S/60 rps
Maximum Rotational Latency = 1 / (S/60 rps)
Average Rotational Latency = Maximum Rotational Latency / 2

Disk Capacity: C bytes
Number of Platters: P
Bytes Per Platter = C/P

Tracks Per Platter Side: T
Sides Per Platter: s
Bytes Per Track = B = (P/s) / T

Bytes Per Sector/Block: b bytes
Number of Blocks/Sectors Per Track = B / b

Cost to Rotate one Block/Sector = Maximum Rotational Latency / (#Blocks/Sectors Per Track) ms

Cost to Rotate N Blocks/Sectors = #Blocks/Sectors to Move * Cost to Rotate one Block/Sector ms

Average Seek Time = Maximum Seek Time / 2 ms

Cost to Seek Single Track = Maximum Seek Time / #Tracks

Seek Time = #Tracks to Move Head / #Tracks * Maximum Seek Time
= #Tracks to Move Head * Cost to Seek One Track

Transfer Time = #Sectors to Read / #sectors per track * Maximum Rotational Latency
= #Sectors to Read * Cost to Rotate one Sector

I/O Service Time = Rotational Latency + Seek Time + Transfer Time