#uni/semester3/Betriebssysteme/chapter21/chapter-notes

Lecture Notes on Operating Systems exercises!
https://www.cse.iitb.ac.in/~mythili/os/ps/memory/ps-memory.pdf

➜ till now we assumed that every address space of every running process fits into memory
➜ in order to support many concurrently running large address spaces, we need to add an additional memory hierarchy.
➜ pages that aren't in demand should be stashed away
➜ this stashing place is served by the hard disk drive (big and slow)
➜ why do we want a large address space for a process? Because you don't have to worry if there is enough space for your programs data structures
➜ in older days **memory overlays** was done, where the programmer had to move pieces of code and data in memory as they were needed.
➜ multiprogramming needs swap space, to hold all pages needed by all processes at once.
➜ so multiprogramming and ease of use are the motives for swapping

1. Swap Space
   ➜ first we need to reserve some space on disk for moving pages back and forth. This space is called **swap space**. The OS can read from and write to swap space in page-sized units. OS will need to remember **disk address** of a page.

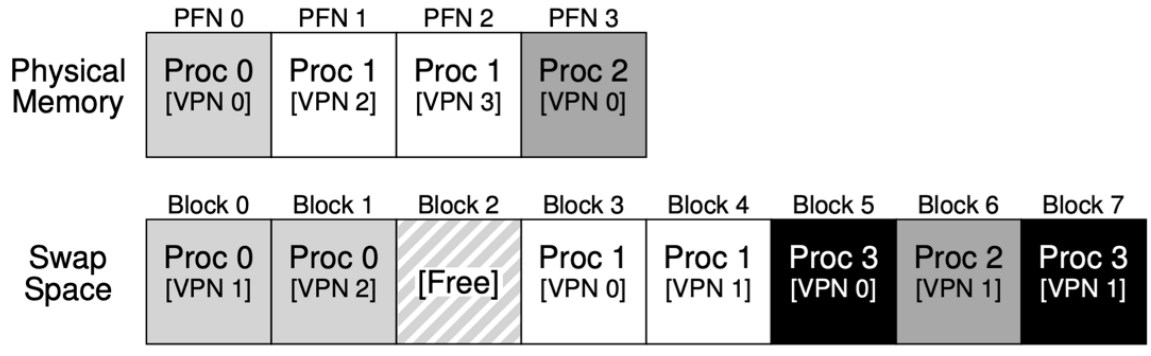

Figure 21.1: **Physical Memory and Swap Space**

➜ three processes are sharing physical memory. Each have some of their valid pages in memory and the rest on swap space.
➜ A fourth process is only on disk and thus is not running.

➜ Swap space isn't the only on disk location for swapping traffic. When you run a program you compiled, the code pages are on disk and when running it, they are loaded into memory (one page at the time when needed).
➜ if system needs to make room in physical memory, it can reuse the pages where the memory space of these code pages, because it knows where to swap them from later.

2. The Present Bit
   ➜ on memory reference, all is like before. Hardware looks in TLB, get it fast if hit, on miss go look in page table. In physical memory, then install translation in TLB and retry.
   ➜ With **present bit** in the PTE the hardware can check if a page is present in physical memory. If 0 the page is not in memory but somewhere on disk.
   ➜ The act of accessing a page that isn't in physical memory is knows as a **page fault**
   ➜ When page fault occurs, the OS comes into play that runs a **page-fault handler**.
   ➜ page fault might also refer to illegal memory access. Therefore a better name for page fault when page is not on memory, is **page miss**. Both on illegal access and page not present, the hardware raises an exception, and transfers control to the OS, which knows how to handle it. That is why both activities are known as fault.

3. The Page Fault
   ➜ both hardware and software managed TLBs, put the OS in charge to handle a page fault.
   ➜ If page not present the OS will need to swap the page into memory.
   ➜ Where is the page in swap space? Information is stored in page table. OS could use the bits in PTE used for data such as PFN to get a disk address. It then issues a request to disk to fetch the page into memory.
   ➜ When disk I/O completes the OS updates the page table to mark the page as present, updates PFN field and retries the instruction. Before retrying the instruction one could also update the TLB to avoid a miss.
   ➜ While I*O process will be in blocked state and OS can run other process. I*O is expensive, the **overlap** of I/O and execution of other process is effective for a multiprogrammed system.
   ➜ Why is the OS trusted to handle the page fault? First faults to disks are so slow, that the extra overheads of running software is minimal. Other reasons are performance and simplicity, because the hardware would have to know how swap space works, how to issue I/O to disk and other details.

4. What If Memory Is Full?
   ➜ What happens when memory is full and we want to **page in** a page from swap space? The OS might want to **page out** one or more to make room for the news ones. **page**

replacement policy is the process to kick out or replace. A good policy is important, because if wrong decision program can run 10 000 or 100 000 times slower.

5. Page Fault Control Flow

```
1   VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2   (Success, TlbEntry) = TLB_Lookup(VPN)
3   if (Success == True)    // TLB Hit
4       if (CanAccess(TlbEntry.ProtectBits) == True)
5           Offset   = VirtualAddress & OFFSET_MASK
6           PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7           Register = AccessMemory(PhysAddr)
8       else
9           RaiseException(PROTECTION_FAULT)
10  else                    // TLB Miss
11      PTEAddr = PTBR + (VPN * sizeof(PTE))
12      PTE = AccessMemory(PTEAddr)
13      if (PTE.Valid == False)
14          RaiseException(SEGMENTATION_FAULT)
15      else
16          if (CanAccess(PTE.ProtectBits) == False)
17              RaiseException(PROTECTION_FAULT)
18          else if (PTE.Present == True)
19              // assuming hardware-managed TLB
20              TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
21              RetryInstruction()
22          else if (PTE.Present == False)
23              RaiseException(PAGE_FAULT)
```

Figure 21.2: **Page-Fault Control Flow Algorithm (Hardware)**

```
1   PFN = FindFreePhysicalPage()
2   if (PFN == -1)              // no free page found
3       PFN = EvictPage()       // run replacement algorithm
4   DiskRead(PTE.DiskAddr, PFN) // sleep (waiting for I/O)
5   PTE.present = True          // update page table with present
6   PTE.PFN     = PFN           // bit and translation (PFN)
7   RetryInstruction()          // retry instruction
```

Figure 21.3: **Page-Fault Control Flow Algorithm (Software)**

➜ The OS must first find a free physical frame

➜ If there is no such page, the replacement algorithm will be run and it will kick out some pages out of memory

➜ With a physical frame ready, the handler then issues an I/O request to read in the page from swap space

➜ After the slow I/O the OS updates the page table and retries the instruction. The retry will result in a TLB miss, because the translation still isn't in the TLB. But the next retry after that will be a TLB hit.

6. When Replacements Really Occur

➜ it is not realistically that the OS waits until memory is full to evict some pages. There are reasons to keep a portion of the memory free proactively

➜ To keep a small amount free, most OS have a **high watermark** (HW) and **low watermark** (LW) that help deciding when to start evicting pages. Describing Physical Memory

➜ When there are fewer pages than LW, a background thread will run that will start to free memory.

➜ This thread evicts pages, until there are HW pages available.

➜ The background thread then goes to sleep. This is also called **swap daemon** or **page daemon**. The word daemon is an old term to describe a background thread or process that does something useful.

➜ By freeing many pages at once, it enables some performance optimisations. For example these pages will be **clustered** or **grouped** together, and swapped to the the swap partition at once. Such clustering reduces seek and rotational overheads of the disk and thus increases performance.

➜ The code from above (21.3) has to be modified to make it work with the background paging thread. Instead of performing a replacement directly, the algorithm would simply check if there are any free pages available. If not it informs the background paging thread, which frees pages that are needed. When it finishes it re-awakes the original thread which will page in the desired page.

➜ the described clustering above has other benefits.

➜ Increases disk efficiency, as the disk may receives many writes at once and has the chance to schedule them better.

➜ improved latency writes, as the app thinks write completed quickly

➜ possibility of work reduction as the writes may not end up to the disk because file was deleted.

➜ utilises the hardware better, because it uses better the idle time. The background work will be done when system is idle.

7. Summary

➜ Every action above takes place transparently to the process. From process POV it just accesses its private, contiguous virtual memory.