17. Swapping : Mechanism

to support many concurrently-running large address spaces
=> additional level in the memory hierarchy

the OS need a place to stash away portions of address spaces that currently aren't in great demand

the characteristics of such a location are that it should have more capacity than memory => generally slower = hard disk drive
*if it were faster, we would just use it as memory

in memory hierarchy, big & slow hard drives sits at the bottom

How to Go Beyond Physical memory?

How can the OS make use of a larger, slower device to transparently provide the illusion of a large virtual address space?

Using more memory than is physically available : Swapping memory

Why do we want to support a single large address space for a process?

: convenience & ease of use
you don't have to worry about if there is enough room in memory for the program's data structures
you just write the program naturally, allocating memory as needed.

in older systems used memory overlays
required programmers to manually move pieces of code or data in & out of memory as they were needed.
Before calling a function or accessing some data, you need to first arrange for the code or data to be in memory

Swap Space

reserve some space on the disk for moving pages back & forth
= memory <-swap pages -> disk

- OS can read from & write to the swap space, in page-sized units
- the OS will need to remember the disk address of a given page
- the size of the swap space is important = determines the maximum number of memory pages that can be in use by a system at a given time
- pages swapped out to disk => isn't currently running

Present Bit

present bit per each page-table entry
/ whether the page presents in Physical memory or not

- present bit == 1 : in memory
- present bit == 0 : in disk (swap space)

add machinery higher up in the system when the hardware looks in the PTE / in order to support swapping pages to and from the disk

Page Fault

TLB miss -> look up page table -> present bit == 0 --> page fault

the OS is put in charge to handle the page fault
OS page-fault handler runs to determine what to do

OS need to swap the page into memory to service the page fault

Why hardware doesn't handle page faults? unlike TLB?
1. no need to speed up software overheads / we use hardware for speed
   : Disk I/O itself is already super slow, optimize software overheads can't do a lot either way
2. too complicated to handle
   : need to understand swap space, how to issue I/Os to the disk, etc...

How the OS know where to find the desired page in the disk?
: the OS could use the bits in the PTE
ex. PFN of the page => for a disk address

While the I/O is in flight, the process will be in the blocked state
the OS can run other ready processes while the page fault is being served
Because I/O is expensive, it's an effective use of its hardware

What if memory is full : page out

page-replacement policy (similar to TLB replacement policy)

similar

TLB <=> memory

memory <=> disk

kicking out wrong page can create severe problem

Making the wrong decision can cause a program to run at disk-like speeds instead of memory-like speeds

```
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()         // replacement algorithm
4  DiskRead(PTE.DiskAddr, PFN)   // sleep (wait for I/O)
5  PTE.present = True            // update page table:
6  PTE.PFN     = PFN             // (present/translation)
7  RetryInstruction()            // retry instruction
```

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

OS keep a small portion of memory free

multiple page fault (I/O)at once => reduce time

High watermark (HW) / Low watermark (LW)

: help decide when to start evicting pages from memory

when the OS notices that there are fewer than LW pages available

1. a background thread (swap daemon*/page daemon) that is responsible for freeing memory runs
2. the thread evicts pages until there are HW pages available
3. the thread then goes to sleep

*daemon / usually pronounced "demon"
term for a background thread or process that does something useful

system can cluster or group a number of pages
& write them out at once to the swap partiton

: such clustering reduces seek & rotational overheads of a disk
=> increase the efficiency of the disk & performance noticeably

Segment vs. Page

Segement : logical, upper level concept
    a segment can have multiple pages
Page : physical, lower level concept

Fault = illegal access

Segmentation fault = invalid

: when a process accesses a memory address **outside of any valid segment**, whether unmapped or violating access permissions defined by segmentation.

Why Page fault is "fault", not "miss"?

: the hardware (dealing process) doesn't know how to handle or can't figure out. toss to the OS by raising an exception

- TLB "miss" : CPU(MMU) <-> Memory
- Page fault : Memory <-> Disk

* disk is not hardware in the light of OS context, since it doesn't handle process (runtime)

Why doesn’t free memory return to the exact same level?

This is due to how **Linux uses memory efficiently**:

1. Memory freed by mem is not immediately shown as "free"

- It may be moved into the **file system cache** or **buffers**.

So technically, the memory is **available** but not counted in the free column.
It becomes part of cached or buffers, which Linux will reclaim if needed.

2. free ≠ usable memory

The free column only shows **completely unused** memory.
But Linux keeps memory **"warm"** (used by cache) to improve performance.

Why swapping might *not* start at first

:In many systems, Linux uses **lazy allocation** — meaning:

Even if you call malloc(6 GB), it won't be backed by RAM or swap **until you touch it**.

So in the **first loop**, you might just be touching memory sequentially — still fits in RAM. You'll see little or no si/so.

take more time for each loop when constanly swappping

1. **Evict some pages** from memory (swap out).
2. **Load needed pages** from disk (swap in).
3. Repeat this process **on every loop**, especially if your program accesses a large memory region in each loop.

The **3rd loop is slower than the 2nd** because:

- More pages are swapped out/in overall.
- Disk I/O contention increases.
- CPU cache and TLB become less effective.
- OS spends more time managing memory.