Chapter 21
Beyond Physical Memory: Mechanism

So far:

- Assumed address space (AS) fits in physical memory (PM)
- Relax this assumption; want to:
  - Support many “concurrently-running” large ASes (process)
  - Support virtual ASes that are larger than PM

Require additional levels in the memory hierarchy

- TLB - Address Translation Cache
- PM - Physical Memory
- Disk (swap space)

To support such large ASes:

- OS need a place to stash away portions of AS
  - That are rarely used
  - This place must have more capacity than PM
    - Slower — recall: the faster-lower capacity vs slower-more capacity tradeoff
    - Typically: HDD or SSD

Q: How can OS use larger, slower devices to
transparently provide the illusion of large virtual ASes?

Q: Why support a single large AS for each process?

- Convenience and ease of use
- Don’t have to worry if there is enough room for you program’s data structures

21.1 Swap Space

Def. Swap Space: *Reserved space on disk for memory pages (page-sized units)*

- Swap pages out of memory to disk
- Swap pages in to memory from disk

OS must keep:

- mapping: memory address to **disk address**

Example. Tiny AS with 4 PM pages (Fig 21.1)

- 4-page PM
- 8-page swap space (SS)
- 3 processes actively sharing PM
  - Each process only have some of their valid pages in memory
  - Rest are located in SS on disk
- Proc 3: all its pages on disk

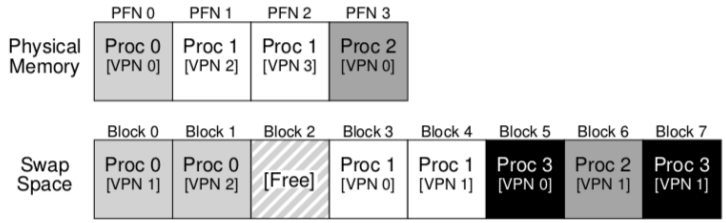

Figure 21.1: **Physical Memory and Swap Space**

21.2 The Present Bit

Recall:

- Process generates a virtual memory reference
- HW translate into physical address — before fetching from memory

HW translation with TLB:

- TLB hit: fast — no additional memory access needed — hopefully common case
- TLB miss (VPN not found in TLB)
  - HW inspect page table in memory (using the **Page Table Base Register**)
    - Lookup Page Table Entry (PTE)
    - If PTE.valid && PTE.present (present in physical memory)
      - Fetch page from memory
      - Update TLB
    - If not PTE.present
      - **Page fault** when page is not present in PM
      - Trigger OS page-fault handler

21.3 The Page Fault (Page Miss)

OS call **page-fault handler** if a page is not present in PM.

- OS must swap page into memory from disk
- How OS know where on disk page is?
  - PT hold such information
  - Use PTE.PFN of page also for disk address
    (replacing the PFN since it is no longer in PM)

When IO completes: We have loaded page from disk into PM

- Update the PT
  - Mark the page as present: PTE.present = true
  - Update the PTE.PFN = new in-memory location of the newly fetched page
- Retry instruction
  - TLB miss: fetch from PT, update TLB
  - Retry the instruction again
    - (Because the instruction is only in PM, not in TLB)
  - Last two steps: could be done as part of the page fault handler to avoid this step!

Before IO completes

- Process is blocked (the process is in the Waiting state)
- OS free to run some other process
  - Making effective use of the HW (CPU)

21.4 What if Memory is Full?

Memory may be full when bringing in pages.

OS may need to **page out** some pages first to make room.

- Picking which page to replace (kick out)
  - **Page-replacement policy** (next chapter)
  - Wrong decision: can cause program to run 10,000x or 100,000x slower

21.5 Page Fault Control Flow

Line 18-23: are new!

Find free physical page!

```
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)**

21.6 When Replacements Really Occur

In reality, the OS proactively keeps a small portion of memory free

Two parameters:

- High watermark (HW)
- Low watermark (LW)

OS runs a background thread: **swap daemon** (or page daemon)

- | free pages | < LW
  - Free memory: evict pages from memory (move to disk)
  - Until | free pages | >= HW
  - Swap daemon goes to sleep

Performing many replacements at once: optimize performance!

Background task:

- Increase efficiency
- Grouping many operations into one (batching)

21.7 Summary

Added present bit in PTE

Page-fault handler:

- Transfer page from disk to memory
- Replace pages to make room for those being swapped in

This all happens **transparently** to you as a developer

- You access your own private, contiguous virtual memory
- Behind the scenes
  - Pages are placed in arbitrary (non-contiguous) locations
  - Fetched from disk
- However, in some cases:
  - Single instruction can take many milliseconds to complete (in the worst case)