Lecture 10: Swapping

Operating Systems

Content taken from: https://pages.cs.wisc.edu/~remzi/OSTEP/

Last few classes

- MMU for translating a virtual address to physical address
- Dynamic relocation with base + bound registers
- Segmentation
- Paging
- Multi-Level Paging
- TLB (cache for virtual-to-physical address translations)

Assumption till now

- Every virtual address space of every running process fits into physical memory.
  - That is, all the pages reside in physical memory.
- Can we relax this assumption?
  - Yes, we can move some of the pages (allocated to processes) to disk
- Why do we need to relax this assumption?
  - To support the illusion of a large virtual memory for multiple concurrently-running processes.

Swap Space

- Reserved space on disk for moving pages back and forth
- OS can read from and write to the swap space, in page-sized units.
  - OS needs to remember the **disk address** of a given page.

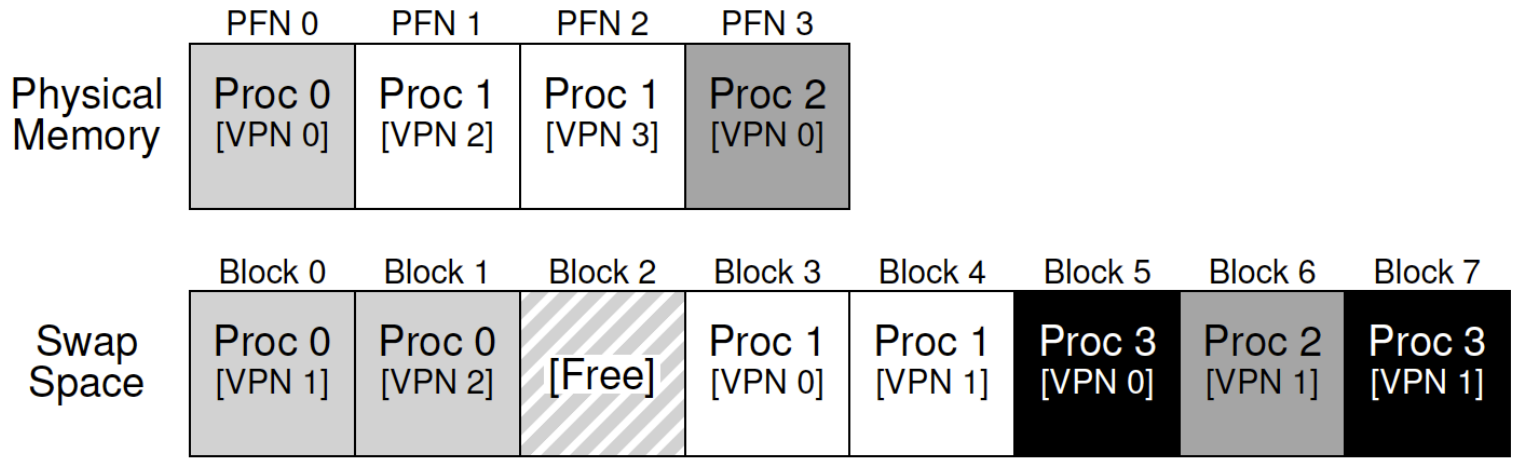

Figure 21.1: **Physical Memory and Swap Space**

Present Bit

- Till now, we have looked at
  - Valid bit
  - Protected bit
- Latest addition: Present bit in each PTE
  - If set to one, then the page is present in physical memory
  - If set to zero, then the page is not present in physical memory but stored on disk

Page Fault

- Trying to access a page that is not present in physical memory
- Upon a page fault, OS is invoked to service the page fault using a **page-fault handler**
- OS finds the disk address for the page in it's PTE.
- OS issues the request to disk to fetch the page into memory
- Once the page is fetched into memory
  - OS updates the PTE to mark the page as present
  - OS updates the PFN of the PTE with the new in-memory location

What if memory is full?

- Replace an existing page with the new pages which need to be brought in
- Page-replacement policy dictates which page to replace?
  - Will talk about it next.

```
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)**

```
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)**

Optimal Replacement Policy

- Main memory can be viewed as a cache for virtual memory pages in the system
- Optimal replacement policy should
  - Minimize the number of cache misses
    - Minimize the number of times we have to fetch a page from disk
  - Maximize the number of cache hits
    - Maximize the number of times a page that is accessed is found in memory
- **Optimal policy:** Replace the page that will be accessed **furthest in the future**

Optimal: Example

- Compute the cache hit rate?
  - #hits/ (#hits+#misses)
  - = 6/(6+5)
  - Page 3 of Textbook chapter URL

Access	Hit/Miss?	Evict	Resulting Cache State
0			
1			
2			
0			
1			
3			
0			
3			
1			
2			
1			

FIFO replacement policy

- Pages are stored in a queue
- Page on the tail of the queue (First-in) is evicted.

FIFO: Example

- Compute the cache hit rate?
  - 4 hits, 7 misses
  - Hit rate = (4/4+7)
  - Page 5 of [Textbook chapter URL]

Access	Hit/Miss?	Evict	Resulting Cache State
0			
1			
2			
0			
1			
3			
0			
3			
1			
2			
1			

Using History: LRU

- If a process has accessed a page in the near past, it is likely to access it again in the near future
- Least-Recently-Used (LRU) policy replaces the least-recently-used page.
  - The more recently a page has been accessed, perhaps the more likely it will be accessed again
- **Least-Frequently-Used** (LFU) policy replaces the least-frequently-used page
  - If a page has been accessed many times, perhaps it should not be replaced as it clearly has some value.

LRU: Example

- Compute the cache hit rate?
  - 6 hits, 5 misses
  - Hit rate = (6/6+5)
  - Page 7 of Textbook chapter URL

Access	Hit/Miss?	Evict	Resulting Cache State
0			
1			
2			
0			
1			
3			
0			
3			
1			
2			
1			

Implementing LRU

- **Option 1:** Maintain a list of pages where the most recently used page is moved to the front of the list on every page access.
- **Option 2:** Update the time field associated with the page just accessed. For replacement, find the page with the oldest timestamp.
- Both options are costly

Approximating LRU

- Store a **use** bit (or **reference** bit) in each PTE
- Whenever a page is referenced (read/written), the **use** bit is set to 1
- How to employ the **use** bit to approximate LRU? [**Clock algorithm**]
  - Imagine all the pages to be arranged in a circular list
  - OS scans through this list and replaces the first page whose use bit is set to 0.
  - Whenever the OS encounters a page whose use bit is set to 1, that use bit is reset to 0.
    - The OS then proceeds until it finds a page whose use bit is set to 0.

Considering Dirty Pages

- If a page has been modified in memory and not yet written to disk, it is marked as dirty using the **dirty** bit.
- The page replacement policy does not evict a dirty page as before eviction, the dirty page necessarily needs to be written to disk which is expensive.
- Clock algorithm can be modified to first consider pages which has both use bit and dirty bit set to 0.

Other policies

- **Demand paging:** OS brings a page into memory when it is accessed i.e. “on demand”
- **Thrashing:**
  - Memory demands of the set of running processes simply exceeds the available physical memory
  - System will be constantly paging (out and in)