50004 - Operating Systems - Lecture 10

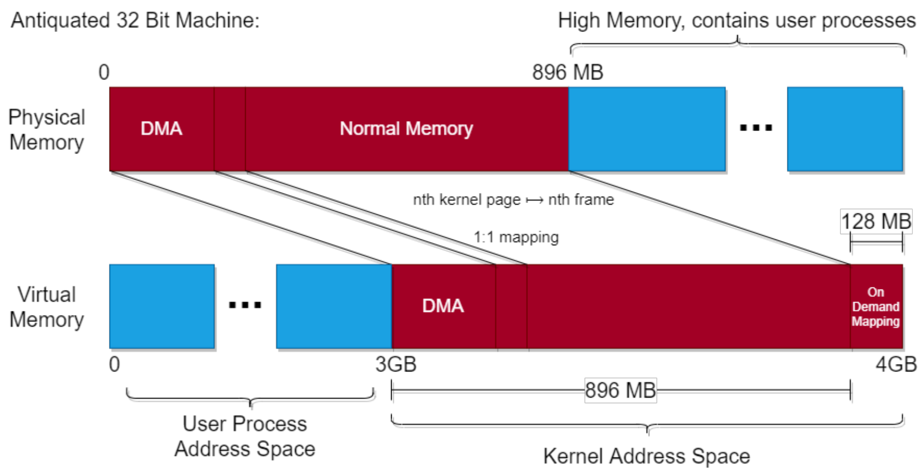
Oliver Killane

22/11/21

Lecture Recording

Lecture recording is available here

Linux - Virtual Memory Layout



- 1 : 1 **Mapping**
  Can turn logical address to physical address for kernel pages by subtracting 3GB.
  - Very efficient for kernel memory access
  - Does not require page table (so TLB is not flushed when user process makes syscall).
  - On Demand Mapping contains temporary mappings for use of more than 896 MB of memory.

Typically on IA32

- 4KB page size.
- 4GB virtual address space.
- Two-level page table (up to 3 with **Physical Address Extension**).
- Offset bits contain page status (dirty, read-only etc).

On AMD64/x86_64

- Larger page sizes (e.g 4MB).
- Up to four-level page table.
- Offset bits can contain can-execute (prevent malicious code being written to take over process).

Meltdown Attack

```
s = a + b
t = s + c
u = t + d
v = u + e
if (v == 0) { /* Context speculativeley executed. */
    w = kernel_mem[addr]
    x = w & 0x1;
    y = x * 4096;
    z = user_mem[y];
}
```

By speculative execution, this becomes:

```
/* ensure user_meme[0...4096] is not in cache*/

s = a + b
t = s + c
u = t + d
v = u + e

/* speculative execution*/
w_ = kernel_mem[addr]  /* no page fault for speculative */
x_ = w & 0x1;
y_ = x * 4096;
z_ = user_mem[y];

/* If statement true, speculative results are set */
if (v == 0) {
    w = w_;
    x = x_;
    y = y_;
    z = z_;
}

/* check how long it takes to read user_mem[0] and user_mem[4096] to determine
 * if it is in cache.
 *
 * If mem[0] has been cached, then kernel_mem[addr]'s LSB is 0
 * If mem[4096] has been cached, then kernel_mem[addr]'s LSB is 1
 */
```

Demand Paging

Only load pages from swap when the user attempts to access them.

- Lower I/O load (unused pages are never loaded)
- Less memory required (fewer pages resident in memory)
- Faster response time (Can start executing straight away, does not need to wait for all pages to be loaded first)
- Supports more users (lower memory usage allows this)

Uses a valid-invalid bit such that:

$$\begin{aligned} 0 &\Rightarrow \quad \text{in memory} \\ 1 &\Rightarrow \text{not in memory} \end{aligned}$$

- All page entries are originally set to 0.
- If a page with bit 0 is accessed, page fault and trap to kernel.
- Kernel uses other table to determine if reference is invalid, or valid but page not in memory.

To handle a valid request the kernel:

1. Get empty frame
2. swap page to frame
3. Reset tables (validation bit = 1)
4. Restart last instruction

Performance

Given a **Page Fault Rate** ($= 0 \Rightarrow Never$, $= 1 \Rightarrow Always$).

$$\text{Effective Access Time} = (1-p) \times \text{memory access time} + p \times \left(\begin{array}{c} \text{Page Fault overhead} \\ + \\ \text{swap page out} \\ + \\ \text{swap page in} \\ + \\ \text{restart overhead} \\ + \\ \text{memory access time} \end{array} \right)$$

Virtual Memory Tricks

- **Copy-on-Write (COW)**
  Processes accessing identical pages use the same frame, only copy when a process wants to write/modify the page.
  - Parent and Child processes can initially share same pages in memory
  - Efficient process creation (copy only modified pages)
  - Free pages allopcated from a pool of zero-ed out pages

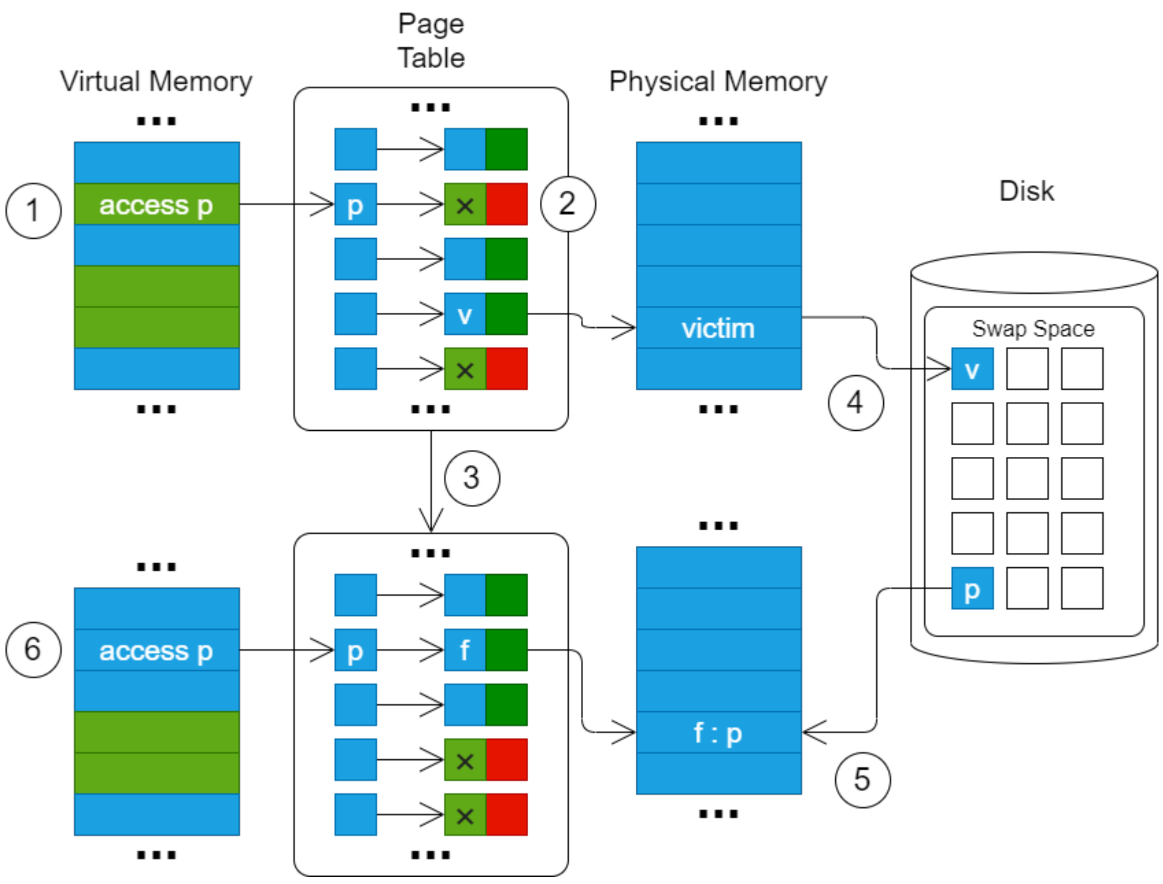
  For example with **fork**:
  - Child's page table points to parent's pages (marked as read-only in child & parent's page table)
  - Protection fault causes trap by kernel.
  - Kernel allocated new copy of the page to process alterning (that it can write to), replaces old page in page table.
  - Both child and parent's page table sets page to Read-Write.
- **Memory Mapped Files**
  Map files into the virtual address space using paging. Only need to load parts of a file when they are accessed.

  e.g 16K page file, only access first 2 pages, so only first 2 loaded to memory.

  Simplified programming model for I/O (can easily access stdin/out).

Page Replacement

When out of free memory & a new page must be created, a page must be swapped out.
An example below (note victim does not have to be a page of the same process).



1. Access a page not loaded into memory.
2. Page table contains bit to show page is not loaded.
3. Update page table (must ensure no race conditions when involving multiple processes that may be running on multiple cores).
4. Write victim to disk (swap space).
5. Read page from disk.
6. Restart operation, can now access page.

The goal is to:

- **Reduce the number of page faults**
  Avoid bringing a page back into memory many times & in general more frames ⇒ fewer page faults.

- **Prevent over-allocation of memory**
  Page-fault service routine should include page replacement.

- **Reduce redundant I/O**
  Use modify (dirty bit) to only load modified pages back to disk.

Replacement Algorithms

FIFO

Replace the oldest page.

Advantages	Disadvantages
• Simple to implement	• May replace a heavily used page.

Belady's Anomaly (where more frames result in more page faults), for example:

3 frames, 9 faults

Access:	1	2	3	4	1	2	5	1	2	3	4	5
Page Fault:	Y	Y	Y	Y	Y	Y	Y	N	N	Y	Y	N
Frame 0:	1	1	1	4	4	4	5	→	→	5	5	→
Frame 1:	N	2	2	2	1	1	1	→	→	3	3	→
Frame 2:	N	N	3	3	3	2	2	→	→	2	4	→

4 frames, 10 faults

Access:	1	2	3	4	1	2	5	1	2	3	4	5
Page Fault:	Y	Y	Y	Y	N	N	Y	Y	Y	Y	Y	Y
Frame 0:	1	1	1	1	→	→	5	5	5	5	4	4
Frame 1:	N	2	2	2	→	→	2	1	1	1	1	5
Frame 2:	N	N	3	3	→	→	3	3	2	2	2	2
Frame 3:	N	N	N	4	→	→	4	4	4	3	3	3

Here the reference string is such that under **FIFO** at 4 frames the next page is frequently the oldest, resulting in a high number of page faults.

Optimal Algorithm

Replace the page that will not be used for the longest period of time. This is impossible in practice, but can be used as a benchmark for comparing other algorithms.

Note that in the example, pages never used again are the longest away & lower frames are chosen when pages have equal time.

3 frames, 7 page faults

Access:	1	2	3	4	1	2	5	1	2	3	4	5
Page Fault:	Y	Y	Y	Y	N	N	Y	N	N	Y	Y	N
Frame 0:	1	1	1	1	→	→	1	→	→	3	4	→
Frame 1:	N	2	2	2	→	→	2	→	→	2	2	→
Frame 2:	N	N	3	4	→	→	5	→	→	5	5	→