CSE 120

Project 3

Project 2 vs Project 3

Processor.numPhysPages = 64

- Userspace access memory directly through page table
- Sufficient physical pages to hold all processes
- Each process's physical pages are loaded into memory and page table mapping is fixed on initialization

Processor.numPhysPages = 16

- Not sufficient pages. Need to swap in/out pages
- No page is loaded at initialization (lazy loading).

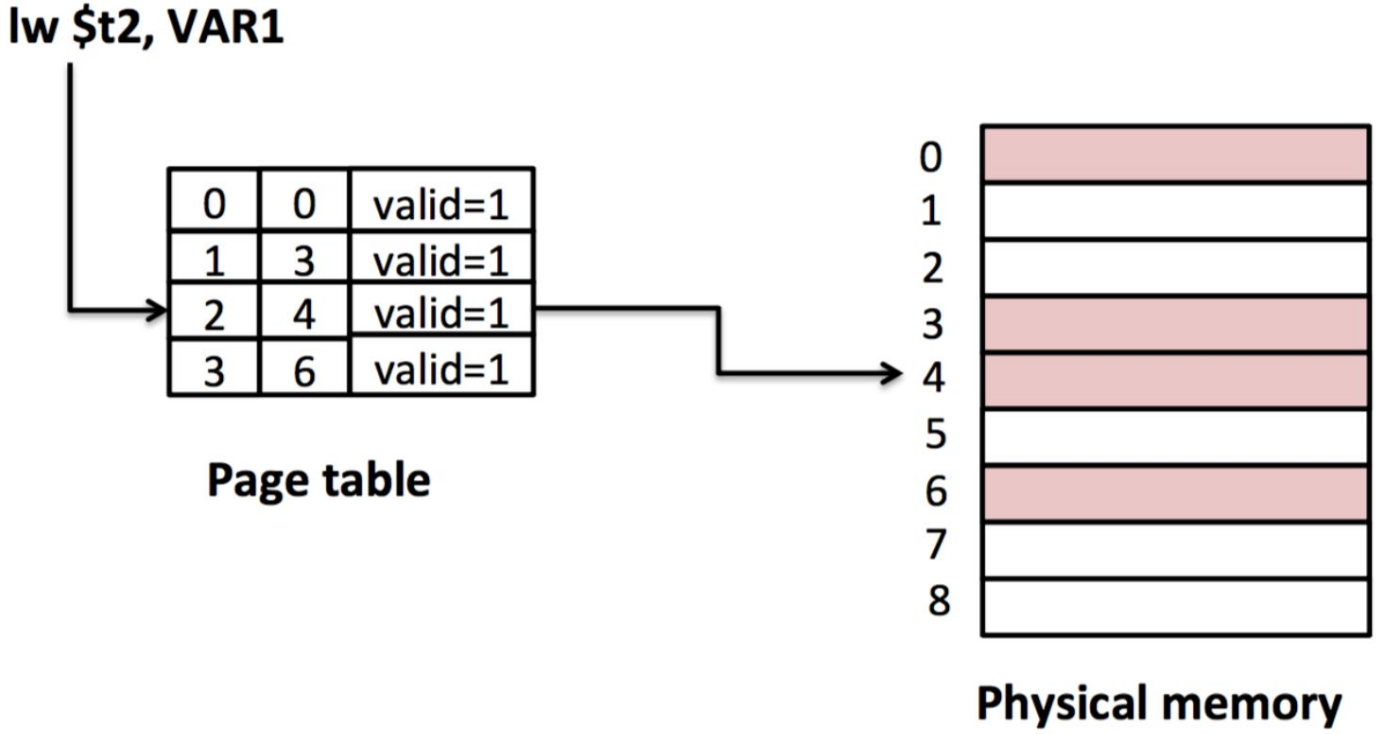
What is “Lazy”

“Lazy” (in computer science) is doing something only absolutely when you need to. For example, “Lazy Iterators”

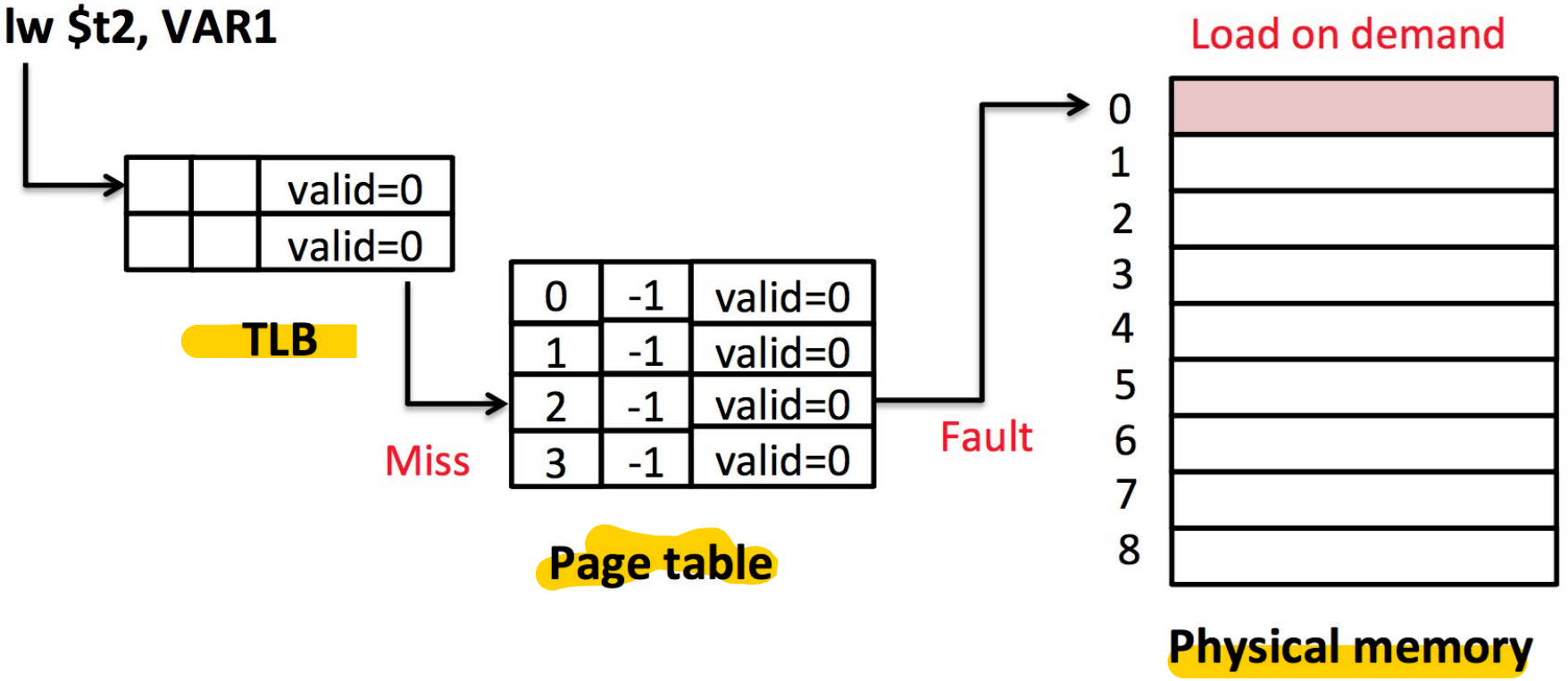
“Lazy loading” would mean loading something when you need that something.

This is fundamentally different than what happened in project 2. You loaded things before you actually needed it in project 2 in loadSections().

Project 2 Recap: Load on Initialization



Project 3: Lazy Loading



Project 2 vs project 3 -- nachos.conf

Machine.stubFileSystem = true
Machine.processor = true
Machine.console = true
Machine.disk = false
Machine.bank = false
Machine.networkLink = false
Processor.usingTLB = false
Processor.numPhysPages = 16
ElevatorBank.allowElevatorGUI = false
NachosSecurityManager.fullySecure = false
ThreadedKernel.scheduler = nachos.threads.RoundRobinScheduler
Kernel.shellProgram = sh.coff
Kernel.processClassName = nachos.vm.VMProcess
Kernel.kernel = nachos.vm.VMKernel

How is it possible run a process when there is less physical pages than it requires??

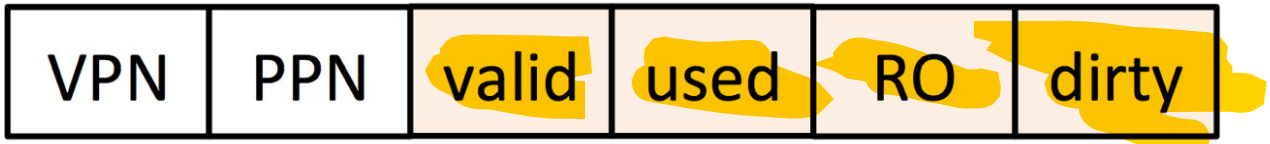
We can use some kind of external storage. We can call this a swap file. We're basically just writing to disk.

When we need more physical pages and there are no more physical pages, we can write the contents of the physical page to the DISK or the swap file. We can then reuse that physical page without losing information about that physical page.

Project 3 Virtual Memory

1) **Demand Paging**. Pages will be in physical memory only as needed. When no physical pages are free, it is necessary to free pages, possibly evicting pages to swap.
2) **Swapping** : No sufficient physical pages for user processes
3) **Lazy loading** : One physical page is ONLY loaded into memory on demand
4) **Page Pinning** : At times it will be necessary to "pin" a page in memory, making it temporarily impossible to evict.

TranslationEntry



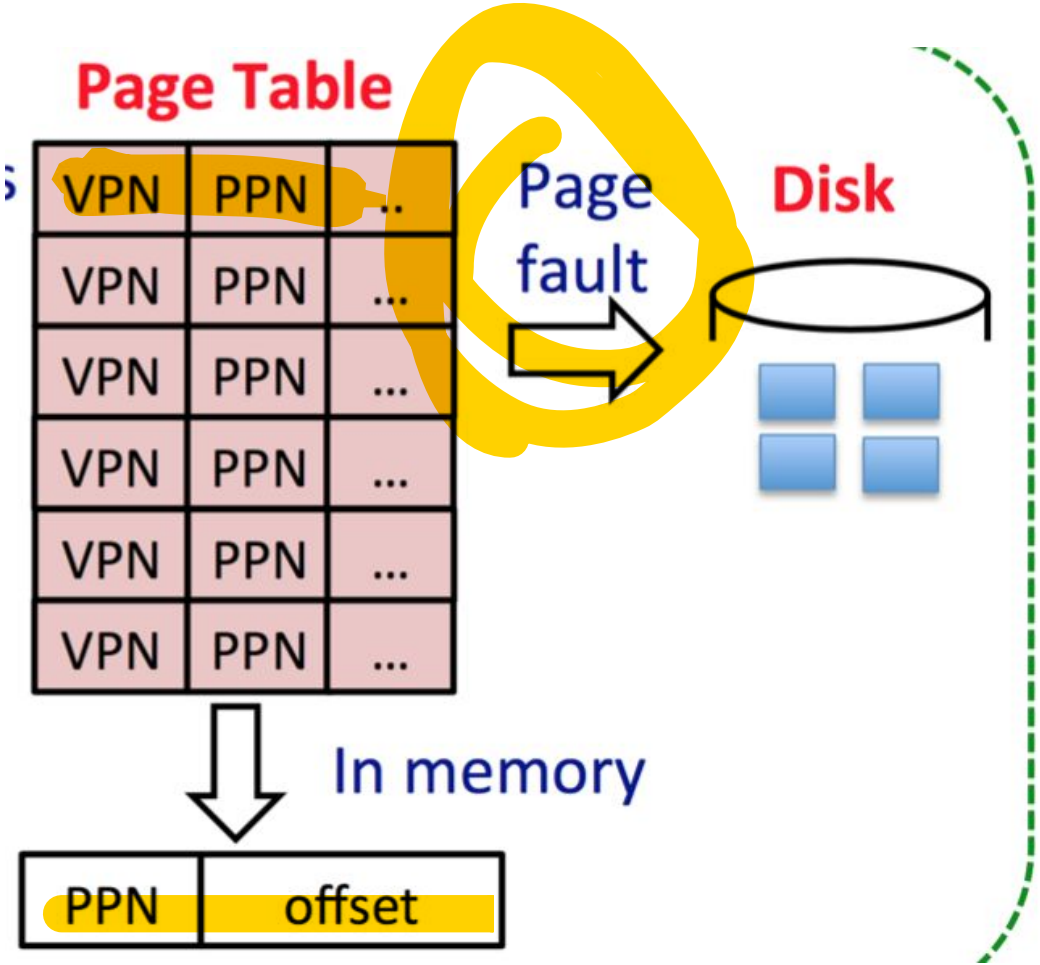
- Valid
  - If the page is in memory.“*false*” invokesTLB miss or page fault
- used(or reference)
  - If the page has been accessed. Important in page replacement.
- RO(read-only)
  - If the page is writable. Important in swapping.
- dirty
  - If the page has been written. Important in swapping

From the highest level, we need to handle page faults.

```
public void handleException(int cause) {
        Processor processor = Machine.processor();
        switch (cause) {
                case Processor.exceptionPageFault:
                        handlePageFault(processor.readRegister(Processor.regBadVAddr));
                        break;
                default:
                        super.handleException(cause);
                        break;
        }
}
```

Whats a page fault exception though?

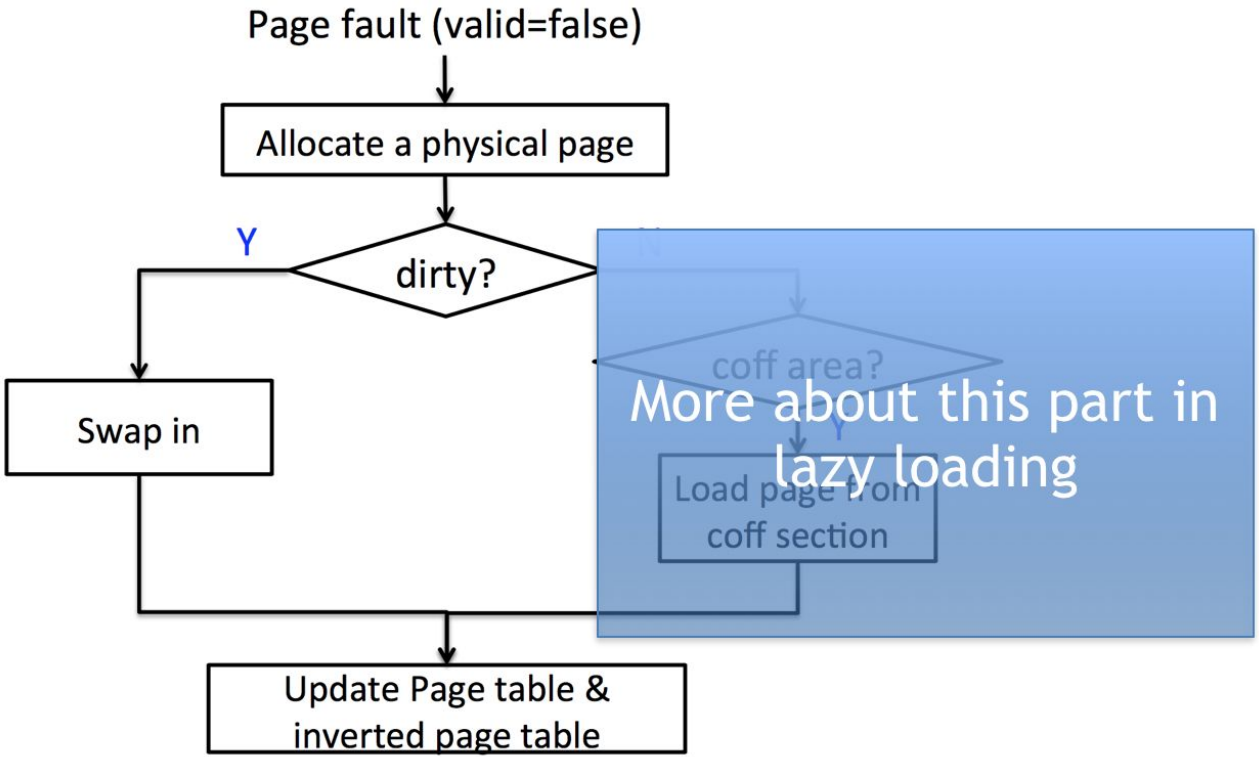
1. When the program is running, it knows what virtual addresses it may need to modify and it uses the PAGE TABLE to look up the physical page that corresponds to that virtual address to do the modification.
2. Remember that each entry in the page table is a TranslationEntry, if the valid bit is FALSE for the translation entry, we get a PAGE FAULT EXCEPTION.
    a. This is a sign that the page is not loaded into physical memory. So we simply have to load it into memory
    b. This is also expected for lazy loading of pages. We expect none of the pages to be loaded into memory at the start because we only load pages when we need them. We know we need them when we get this page fault exception.


Page Table
VPN
PPN
..
VPN
PPN
...
VPN
PPN
...
VPN
PPN
...
VPN
PPN
...
VPN
PPN
...
Page fault
Disk
In memory
PPN
offset

High-Level Behavior

- Handling page fault
  - Select a physical page in which to load the virtual page
  - Need to load the required page to memory (from swap file or .coff sections)
- Swapping
  - Swap out : write page content to swap file
  - Swap in: read page content from swap file to memory

Handle Page Fault



Update PageTable and Inverted Page Table

Okay so I know what a page table is. I played around with it for project 2.

But what is an inverted page table?

Why do we need it?

Remember the VALID bit in a TranslationEntry corresponds to whether the page is in physical memory. Pages can now leave physical memory and the page table of a process needs to be aware that page no longer “belongs” to them.

Algorithm for Page Allocation

```
if (freePageList is not empty)
{
    Allocate one from list; // same with proj2
}
else {
    //No free memory, need to evict a page
    Select a victim for replacement; //Clock algorithm
    if (victim is dirty) {
        swap out;
    }
    Invalidate PTE and TLB entry of the victim page
}
```

Inverted Page Table

- Indexed by physical page number
- Used to correlate the physical page with its process and page table entry information
- When you update an entry of the page table, you need to update the corresponding entry in IPT too

ppn -> process

Swapping

- When to swap out?
  - If the page to be evicted is dirty
- When to swap in?
  - If the accessed page has been swapped out

More Swapping

- How to manage swapped pages on disk ?
- How to allocate/reclaim swap space ?
- How to associate the swapped page with the process's address space?
- Which page should/shouldn't be swapped out ?

How to Manage the Swap File?

- Create swap file on VMKernel initialization
  - `File = ThreadedKernel.filesystem.Open(swapFileName)`
- Read/write swap file for swap in/out
  - `File.read()/File.write()`
- Delete swap file on VMKernel termination
  - `File.close()`
  - `ThreadedKernel.filesystem.remove()`

Using the Swap File

- In project 2, we use a global free page list to allocate/ reclaim physical pages
- Similarly, we can just treat swap file as a free page list on the disk
  - The unit of swap file is a page.
  - Each item in the list is the page number in the swap file
    - Similarly with ppn stored in the free page list
  - The swap file can safely grow arbitrarily

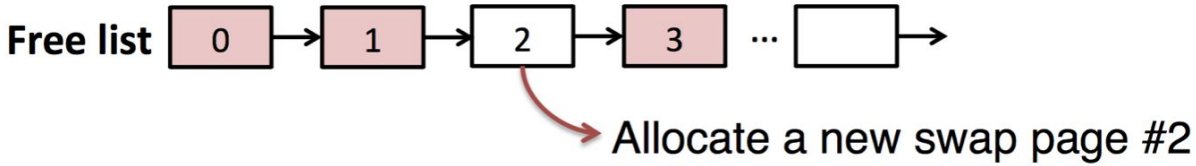
Keeping Track of SwapFile

1. Swap out 5 pages
2. Swap in 2 pages
3. Swap out 1 page

So we need to record which places are available in the swap file

```
private static LinkedList freeSwapPages
```

Reading/Writing from/to SwapFile



- Now we've already recorded the location of a swapped page in the swap file, e.g.
  - Allocate swap page with index 2 from the list
  - Write/read the page content to/from physical page frame to the swap file
    - File.write(2*pageSize, memory, paddr, pageSize);
    - File.read(2*pageSize, memory, paddr, pageSize);