CIS501: Virtual Memory

Shreyas S. Shivakumar

December 10, 2019

1 Lecture Notes

1. Introduction

The *Operating System* is responsible to manage all the different hardware in a computer system. This encompasses a variety of tasks that run in the background, invisible to the programmer.

The *Operating System* **virtualizes hardware** for applications by providing *Abstraction* (threads, files, etc.) and *Isolation* (illusion of private CPU, memory, I/O).

2. Virtualization of a Resource

- To *virtualize* a resource is to make a *finite* amount of a resource act like an infinite or *very large* amount.
- It is then easier for a programmer to write programs with a virtualized interface.
- The resources that are often virtualized are processors, DRAM and sometimes even an entire machine and operating system.

Note: Virtualization can be thought of as creating a *pretend* resource in place of the original one.

3. How do you virtualize a processor?

The goal is to give every program running on the processor the illusion that they are all running on individual processors (*pretend processors*). The *operating system* then needs to be able to switch between the different applications fast enough that the user / application isn't aware that the processor is actually being shared by multiple programs.

This switching between applications is called **Context Switching**.

If **no state** is involved, this is pretty simple : just flush the pipeline and get rid of any instructions that were in the middle and then start up the new program. This isn't too expensive.

When **state** is involved, it needs to be saved somewhere and then restored. For now, let's think of the state as the *register files, registers, PC.*

4. When to perform a *Context Switch*?

A simple approach is to use a *timer* and each process gets fair access to the CPU. It's more efficient not to rely on programs to voluntarily release the CPU.

5. Virtualizing Memory

A naïve solution is to take the memory state and store it somewhere.. (*but where?*, you are using memory.. *maybe disk?* that's too slow!).

Approach: Instead of trying to take memory state and put it somewhere else (this is very slow), everybody gets a *slice of memory.* Even when you are stopped, that memory is still assigned to you and with some clever book-keeping we can prevent other processors from accessing your slice of that memory. Each slice must stay in place and not get trampled on.

The subset of that memory that is actually being used can be moved to **DRAM** and it will behave like a *cache.* The *Operating System* will manage this caching. If something spills out of main memory, it can be pushed to disk.

Note: **Lots of book-keeping involved!**

Note: We want to take pretend memory and multiplex it onto some finite amount of actual memory.

6. Virtual Memory (VM)

- Each program has access to some amount of virtual memory (*pretend memory*). Each program is given the illusion that it has access to a large amount of memory using a level of indirection.
- For *example:* a 64 bit system will have a virtual memory of 2^{64} Bytes in size!
- Programs will then access / reference these *virtual memory locations*. But we will then need to **translate** this to the actual *physical memory locations*.
- This *translation* is done at **page granularity**. This is usually much larger than *byte granularity*. *Pages* are of the order $4\ KB - 64\ KB$.
- Every instruction will generate at least one *virtual* to *physical* address translation.
- *Pro:* You can write your program once and it can be ported to newer machines that are getting bigger and bigger (larger cache capacities).
- *Pro:* Virtual Memory isolates programs from one another.
- *Pro:* Allows multiple programs to run at the same time. Also provides additional *security*.
- *Pro:* Multiple programs can communicate with each other using virtual memory.

7. Address Translation

Virtual Address to **Physical Address** mapping is called *Address Translation*. The Virtual Address is split into *Virtual Page Number (VPN)* and *Page Offset (POFS)*. The **VPN** is translated to *physical page number (PPN)*. The *POFS* is not translated. $[VPN, POFS] \rightarrow [PPN, POFS]$

Most *Operating Systems* will implement this with a structure called a **Page Table (PT)**. It maps Virtual Pages to Physical Pages or to Disk addresses. The actual translation process is a *table lookup*.

Address Translation Example: How big is the page table on a machine that is a 32-bit machine with 4B page table entries (PTEs) and has 4KB pages.

Since it is a 32-bit machine, it has 32-bit Virtual Addresses that is equivalent to $2^{32}B = 4GB$ of virtual memory.

Number of virtual pages $= \frac{\text{Virtual Memory}}{\text{Page Size}} = \frac{4GB}{4KB} = 1\ Million\ VPs$

Since there are 1 Million VPs which are 4 Bytes each this is 4MB. This is then given to every process!

- If you increase the size of the pages, the size of the table will *reduce*. Say you move from 4KB pages to 64KB pages.
- If you move to a 64-bit machine, this increases drastically. $2^{64}B = 19\ Exabytes$. This is **too large**.

One solution is to use *Multi-Level Page Tables (PT)*. Virtual Page numbers are split into two halves which map to different intermediate tables. The first half indexes one table, whose value is the index to the next table, essentially providing two levels of *indirection*. This can be generalized to many levels. This results in a *sparse* amount of memory since a *large amount of virtual addresses* are **unused**.

8. Translation Mechanics

Q: Why a Translation Lookaside Buffer (TLB)?

Doing this page translation process every single time can be very slow. **Translation Lookaside Buffers (TLBs)** are small caches that can help speed this up by caching translations. Instead of *walking the page table* before very $LOAD$, $STORE$ or $FETCH$, only *walk the page table* on a $TLB\ Miss$.

Q: Where does the TLB live?

Conceptually, the *Translation Lookaside Buffer (TLB)* lives between the processor pipeline and the caches. Virtual Addresses come out of the processor, they go through the TLB and Physical Addresses come out of the other side. This indicates a sort of sequential / serial step between the CPU and the Caches.

Q: Can this be improved further?

By treating it as a serial operation that exists between CPU and Cache, it ends up adding *one cycle* to t_{hit}. A clever trick is to access the TLB in parallel. This is possible as long as the VPN and *index* fields don't overlap. This is because the *index bits* don't change since the *Page Offset (POFS)* doesn't change between virtual and physical addresses, so you can go ahead and start the TLB access in parallel. Once you have the Physical Address then you can use the *tag* field but that can come later. The only constraint is that **index bits cannot overlap with VPN bits**.

$$\frac{\text{cache size}}{\text{associativity}} \leq \text{page size}$$