Operating Systems: Shortnotes

Hardik Rajpal

October 1, 2023

Contents

Chapter 1

Introduction

Chapter 2

CPU Virtualization

Terms

- Process Control Block: A per-process data structure that tracks process meta-data.

Chapter 3

Memory Virtualization

Terms

1. Address Space: The running program's view of the memory available to it. Every program only sees the memory available to it, which is split between the code, (static) data, heap and stack.
2. Sparse Address Space: An address space where most of the memory between heap and stack is unused.
3. Memory Management Unit: The hardware responsible for V2P translation.
4. Protection Bits: Bits in VA that represent the process' access permission for the PA to which it points, including read/write/execute permissions.
5. Fragmentation: Wastage of physical memory space.
   - Internal: Wastage within a contiguous block of physical memory allocated to a process. An example is the unused space between heap and stack in a simple base-bounds V2P map.
   - External: Wastage outside of contiguous blocks that have been allocated. This is because of the gaps of available physical memory being too small to fit any new contiguous segments.
6. Page: Unit of address space.
7. Page Frame: Unit of physical address space to which a page can be mapped. Same unit, but **pages** exist in the virtual address space whereas **page frames** exist in the physical address space.
8. Spatial Locality: nearby instructions (in the control flow) access nearby memory locations.
   Ex. with code for sequential array processing.
9. Temproal Locality: nearby instructions (in the control flow) accessing a single memory address repeatedly.

3.1 Address Space

The program is actually loaded into random physical addresses, but due to the address space abstraction, the program sees the memory available to it as a contiguous chunk, starting at 0. Every address the program sees is virtual and the OS (with some translation mechanism of the VA to PA) uses the PA whenever the program references the VA.

3.2 Address Translation

All address translation is hardware-based for efficiency. Each memory access (load/store) is intercepted by the hardware and the VA is translated to a PA. The OS helps setup the hardware for the right translation (per process), and manages memory.

3.2.1 Dynamic Relocation a.k.a. (Base, Bound)

- Two registers for this in the CPU: base and bound.
- Translation:

```
V2P(VA v){
    if(v > bound || v < 0){
        throw "Out of Bounds";
    }
    return (base + v);
}
```

- Values of base and bounds for each process are stored in its PCB.
- Base and bound registers are privileged: if access is attempted from user mode, the OS terminates the access-requesting process.

3.2.2 Segmentation

- Generalized base and bounds for each logical segment of each process: code, heap and stack.
- Maintain base and size for each segment:

Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

- Use VA[0:2] to represent the segment type.
- Better handles **sparse address spaces**, where the program often has very little heap and stack data and thus a lot of the in-between space in a contiguous allocation is wasted.
- Translation is more involved:

```
V2P(VA v){
    segment = v[0:2];//or bit operations.
    offset = v[2:];//or bit operations.
    if(offset < 0 || offset > segdata[segment]["bound"]){
        throw "Out of Bounds"
    }
    else{
        return segdata[segment]["base"] + offset;
    }
}
```

- OS responsibilities:
  - On each context switch the **segmentation table** for the process is replaced by the incoming process' table.
  - On a receiving new process (and its accompanying address space), the OS has to find space in the physical memory for its segments. If the segments are of varying sizes (bounds) this is more involved.
  - Variable size segments lead to external fragmentation.
  - A solution to external fragmentation is periodic compaction:
    1. Stop running processes.
    2. Copy their data to a contiguous chunk of memory.
    3. Update their segment register values.

This creates available contiguous chunks of physical memory but compaction is expensive (copying segments is memory-intensive) and would take up a fair amount of CPU time.

– Alternative approaches involve using a free-list management algorithm like:
  * best-fit
  * worst-fit
  * first-fit
  * buddy algorithm
– External fragmentation will always exist in this scheme, however. The algorithms above simply aim to minimize it. The real solution is to disallow variable sized segments.

- Some systems merge code and heap segments to use only one bit to represent segment.
- There are **implicit** approaches to identify the segment too, where the program infers the segment by identifying how the VA was conceived:
  – from a PC $\implies$ use code segment.
  – from ebp $\implies$ use stack segment.
  – else, use heap segment.
- We also need an additional bit to identify the direction of growth of the segment, if this varies across the segments. The translation function is modified to take this into account.

Sharing Segments

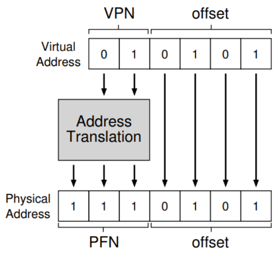
- With some VA bits used to represent permissions that a process has for the segment **protection bits**, we can factor include segments with varying permissions.
- Translation function is appropriately modified to raise an exception whenever a process attempts to violate permissions for a PA.
- Read-only code can be shared across multiple processes, without the worry of harming isolation.

Fine-grained Vs. Coarse-grained

- Segments of code, heap and stack are relatively large (coarse-grained segmentation).
- Fine-grained segmentation further splits up each of these logical segments and identifies each segment using a **segment table**. This allows for better memory management by the OS (by moving unused segments to disk).

3.2.3 Paging

- Split the address space (abstraction visible to the process) into fixed-size units called pages. The corresponding contiguous units in the physical address space are called page frames.
- To record each page's mapping for each process, the OS maintains a **page table** per process that is pointed to by the **Page Table Base Register (PTBR)**.
- Translation: (neglecting bits other than VPN and offset)



Inside The Page Table

- The OS stores the page table in memory, in units of **page table entries** (PTEs) because of the sheer size of page tables.
- Each PTE has a physical address to which it maps and a collection of metadata bits.
- The PTEs are stored in a (nested) array-like structure, such that a VPNs can be used as indices into the structure, pointing at the PTE that contains their corresponding PFN.
- The metadata bits:
  1. Valid bit:indicates whether PTE is valid.
     - When a program starts running, all the (virtual) space between the heap and stack is unused and hence unmapped. Hence entries at pagetable [VA] for those addresses are invalid.
     - If a program tries to access invalid memory, a trap is generated and the program is likely terminated.
     - The valid bit is crucial for supporting **sparse address spaces**.
     - Invalid PTEs don't have any physical memory mapped to them, this lets us use only the physical memory we need.
  2. Protection bits: indicate whether the page can be read from, written to or executed from. Invalid accesses generate a trap to the OS.
  3. Present bit: indicates whether page is present in memory (or has been swapped out to disk).
  4. Dirty bit: indicates whether the page has been modified since being loaded into memory.
  5. Reference (a.k.a. accessed) bit: used to track if the page has been accessed since the bit was last reset. This information helps in page replacement, to keep popular pages around.
  6. User/Supervisor bit: indicates if processes can access the page in user mode.
  7. Hardware caching bits: help determine how hardware caching works for the page.

How It Works

When the OS reads a command to access/write to a memory location at `VA0`,

1. Extract `VPN` and `offset` from `VA0`.
2. Access **PTE** stored at `PTBR + VPN`. **Memory Access 1 (PTE read)**
3. Extract the `PA` and **permission bits** from **PTE**.
4. Verify **permission bits**.
   - If page is invalid, raise `Segmentation Fault` exception.
   - If permissions are insufficient, raise `Protection Fault` exception.
5. Read/write to data stored at `PA + offset`. **Memory Access 2 (Data access)**

Making Paging Faster

The procedure given above is without the use of TLBs, which are always used in the real world to speed up paging.

Reducing The Page Table's Size

1. Increase Page Size: the factor by which the page size is increased is the factor by which the page table's size is reduced.
    - This approach is commonly used by DBMSs and other high-end commercial applications. However, their goal is to reduce pressure on the TLB by reducing the number of translations, instead of minimizing the page table's size.
    - The major problem with this is internal fragmentation. Hence, it's only suitable if the programs are guaranteed to use the large pages allocated to them almost thoroughly.
2. Hybrid approach: Paging + Segments
    - We first note that in each table, most of the PTEs are invalid, and only a select few are mapped to physical page frames.
    - This approach involves maintaining a separate page table for the code, stack and heap segments of the virtual address space.
    - We have base and bound registers for each segment as before, except that the **base** points to the physical address of the page table of that segment.
    - Note that all six registers must be updated on a context switch.
    - The translation involves:

    ```
    V2P(VA v){
        segment = v[0:2];//or bit operations.
        VPN = v[2:k];//or bit operations.
        offset = v[k:];//or bit operations.
        addressOfPTE = Base[segment] + (VPN*sizeof(PTE));
        PTE = read(addressOfPTE);
        PFN = PTE[:c]
        PA =  PFN + offset;
        return PA;
    }
    ```

    - However, like segmentation, (i) this method can't help with the wastage of memory due sparse usage within segments (ex. a sparsely used heap), and (ii) leads to external fragmentation, as the variable sized page tables need to be allocated contiguous chunks of memory.
3. Multi-level page tables:

Advantages of paging

- Flexibility is an important advantage of paging: it works regardless of how the process uses the address space, how the heap and stack grow or how they are used.
- Flexibility also ensures it can handle sparse address spaces and minimize internal fragmentation.
- Simplicity: when new pages are requested, the OS simply traverses a free list and returns the first pages that it finds.
- It avoid external fragmentation altogether by having fixed-size units.

3.2.4 TLB

- It's a part of the MMU and is a hardware cache of popular V2P translations.
- The process of memory access with the TLB is:
  1. Extract `VPN` and `offset` from `VA0`.
  2. If TLB has an entry for `VA0` (TLB hit)
     - If permissions are insufficient, raise `Protection Fault` exception.
     - Extract `PA` for `VA` from TLB.
     - Read/write to data at `PA + offset`. **Memory Access 1 (Data access)**
  3. Otherwise, carry out steps 2, 3 and 4 of the usual access routine. **Memory Access 1 (PTE read)**
  4. Instead of step 5, insert the translation `(VPN,PA,Protection Bits)` into the TLB.
  5. Retry the memory access instruction (start at 1 of this sequence).
- Like most caches, TLB is built on the premise that most translations are found in the cache (hits). The little overhead costs of using a TLB (little because it's near the processing core and designed to be quite fast), are made up for by the hit cases.
- The speed of a cache comes partly from its minimal size; any large cache is by definition slow.

TLB Misses

- We say a program exceeds the TLB coverage if it generates a large number of TLB misses, by accessing (in a short period of time) more pages than the number of translations the TLB can fit.
- One solution to this is to allow larger pages for storing key data structures that such programs access repeatedly, and thus allowing a single translation in the TLB to serve many memory accesses.
- Support for large pages is often exploited by programs such as DBMSs, which have large and randomly accessed data structures.
- In CISC computers, the hardware would handle the TLB miss entirely (low trust on the OS designers).
- With physically-indexed caches, some translation of the VA has to take place for the lookup itself, and thus they are very slow. Check cache notes if any.
  On a miss,
  - The hardware would walk the page table using `PTBR` to get the `PA` for the `VPN` that induced a miss.
  - It would update the TLB with the new translation.
  - Then hit retry on the instruction that resulted in a TLB miss.
- In RISC computers, the TLB is software-managed. On a TLB miss, the hardware just raises an exception (`TLB_MISS`).
- With software-managed TLBs, it is important that the instructions provided by the hardware to read/write to the TLB, are only allowed to be run in privileged modes.
  1. This pauses the current instruction stream, raises the privilege level to kernel mode and jumps to a **trap handler**.
  2. The targeted code looks up the translation in the page table, uses **privileged** instructions to update the TLB, and returns from the trap.

  Note that:
  - Note that on return the execution must pick up **at the causal instruction** that caused the trap, instead of the usual return-from-trap where execution picks up **after the causal instruction.**

- While accessing the code for trap handler for `TLB_MISS`, the OS should not incur a TLB miss, which would imply an infinite chaing of TLB misses. To avoid this, we can:
  1. Keep TLB miss handlers in **unmapped** physical memory, so that their execution does not involve address translation (and hence TLBs).
  2. Reserve entries in TLB for permanently-valid (**wired**) translations and use some of these slots for the trap-handler code's memory address' translation.

- The advantage of a software-managed TLB include:
  - Flexibility: the OS can store the PTEs in any data structure it wants, as it's allowed to update the page table walk function, since it's not burned into a chip.
  - Simplicity: the hardware just has to raise an exception and relies on the OS to resolve the TLB miss.

TLB Contents

- The TLB has 32/64/128 entries and is **fully associative** ($\equiv$ any translation can be anywhere in the TLB, and the hardware searches the entire TLB in parallel to find the entry for `VPN`).
- The TLB entry has: VPN | PFN | other bits.
- The other bits are:
  - Valid bit: denoting if **the TLB entry is valid, NOT the page.**
  - Protection bits.
  - Global bit, to indicate globally shared translations. If this is set, ASID bit is ignored.
  - Address-space identifier (ASID) (to identify if the entry is for the given process or in a shared region).
  - Dirty bit, to indicate if the page has been modified since it was brought into memory.
  - Coherence bits: they determine how a page is cached by the hardware.
  - Page mask field: to support multiple page sizes.
- On a context switch, the OS must ensure one processes translations in the TLB are not used by another process (unless it's a shared memory region). This can be done via:
  1. Simply flushing the TLB on a context switch (marking each entry as invalid). This can be coded up in the OS or the hardware can be set to flush the TLB whenever the PTBR is updated.
  2. Maintain ASID for each entry and a privileged register to hold the ASID of the current process. Note that ASIDs require much fewer (8ish bits) as compared to PIDs (32ish bits).
     Additionally, ASIDs (along with the permission bits) enable sharing of code pages across processes.

Replacement Policy

We want to maximize the hit-rate in the TLB based on our choice of a replacement policy. Some common choices are:

1. Evict Least Recently Used entry: relies on temporal locality of address usage. The least recently used entry in the cache will probably not be used any time soon, as future access will be closer to the more recent accesses in the cache.
2. Random: Avoids making corner-case worst-possible decisions all the time. (For example, a for loop which requires TLB eviction, with the LRU policy in place.)

3.3 Free Space Management

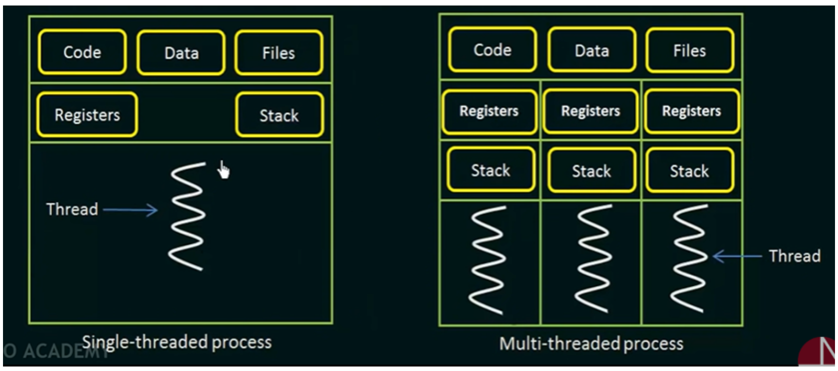
Chapter 4

Concurrency

From Neso Academy's OS lectures.

4.1 Basics of Threads

- Each program may have multiple processes associated with it. Each process may have multiple threads; threads are a basic unit of execution or CPU utilization.
- A thread comprises of:
  1. A thread ID
  2. A program counter
  3. A register set
  4. A stack.
- Threads of the same process share:
  1. Code section
  2. Data section
  3. Other OS resources, such as files and signals.
- A **traditional/heavyweight** has a single thread of control. Having multiple threads of control allows a process to perform multiple tasks at a time.



- The benefits of multithreaded programming are:
  1. **Responsiveness**: interactive applications allow a program to continue running even one of its tasks is blocked in its thread in a lengthy operation, thereby allowing responsiveness.

2. **Resource Sharing**: Code, data and files are shared between threads, allowing for better utilization of system resources.
3. **Economy**: Saves the costs of allocating a separate process for each task.
4. **Utilization of Multiprocessor Arch.**: Multithreading on a multi-CPU machine increases concurrency by using parallelism.

4.2 Multithreading Models and Hyperthreading

- There are two types of threads:
  1. User threads: Supported above the kernel and are managed without kernel support.
  2. Kernel threads: Supported and managed directly by the OS.
- Multithreading models describe the type of relations between user and kernel threads. There are three common types:
  1. Many-to-One
  2. One-to-One
  3. One-to-Many

4.2.1 Many-to-One

- Many user-level threads are mapped to a single kernel thread.
- Thread management is done by the thread library in user space; thus, it is efficient.
- The limitations are:
  1. The entire process will block if a thread makes a system call.
  2. Since only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiple processors.

4.2.2 One-to-One

- A bijection between user-level and kernel-level threads.
- Provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- Also allows multiple threads to run in parallel on multiple processors.
- The limitations are:
  1. Creating a user thread requires creating a corresponding kernel thread, which is costly.
  2. The overhead of creating kernel threads can burden the performance of an application; most implementations of this model restrict the number of threads supported by the system.

4.2.3 Many-to-Many

- Each user thread is mapped to many shared kernel threads. # kernel threads $\leq$ # user threads.
- The number of kernel threads may be specific to a particular application or machine.
- Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
- Blocking system calls result in the kernel scheduling another thread for execution.
- This model makes the best of the previous two models and is what is implemented in most systems.
- However, with its difficulty of implementation and increasing number of cores, one-to-one is increasingly preferred.

4.3 Hyperthreading a.k.a. Simultaneous Multithreading (SMT)

- Multiple multithreaded programs running at the same time.
- Hyperthreaded systems allow their processor cores' resources to become multiple logical processors for performance.
- In hyperthreading, physcial cores are logically divided into multiple virtual cores, to support multiple threads running concurrently.
- It enables the processor to execute two (or more) threads, sets of instructions, at the same time.
- Hyperthreading allows two streams to be executed concurrently, it s like having two separate processors working together.

4.4 Notes from Interview Questions

Chapter 5

Misc.

Chapter 6

References

1. OSTEP