Virtual Memory

Shamelessly taken from
6.033 Course Note Chapter 5.A

Client/Server Architecture

- Limit interactions between modules to messages.
- Good for:
  - Modularity
  - Fault tolerance
  - Security and Protection

Client/Server Architecture (cont.)

- New design opportunities:
  - Multiple clients, multiple servers
  - Buffered communication
  - Sharing resources with people you do not trust.

Client/Server's Big Problem

- Each module has to be in one computer.
- This is very costly and unreasonable.
- Need to pack several modules into one computer.

Virtualization

- Fool each module that its own computer.
- Three abstractions needed to be virtualized:
  - Virtual processor
  - Virtual memory
  - Virtual communication channel
- All of this are handled by the OS.

Virtual Memory

Memory Abstraction

- Two Operations:
  - READ(address)
  - STORE(address, value)
- In systems nowadays:
  - Addresses are 32-bit or 64-bit numbers.
  - Each address refers to a byte.
- So, memory can be viewed as a 2^{32} or 2^{64} contiguous array of bytes.

Why Virtual Memory?

- Without it, modules read and write directly to physical memory.
- A module might STORE invalid data on top of other modules’ data.
- A module might jump into the code of other modules.
- The system is more likely to break if one module screws up.

Virtual Memory

- Fool every module that it has memory address 0 to 2^{32} for its own exclusive use.
- A module can only read/write from/to its own virtual memory.
- A module can only jump to instructions in its own virtual memory.

Virtual Memory (cont.)

- In effect, it is a layer of indirection over physical memory.
- A module uses **virtual addresses** to interface with memory.
- The system **translates** these virtual addresses to **physical addresses**, which is used to interface with the memory hardware.
- The translation process is done by the **virtual memory manager**.

Translation

- This virtual address maps to which physical address?
- Dumb approach
  - Keep a table that maps each virtual address to the corresponding physical address.
  - Doesn’t work because of the heavy memory requirement.
- A better approach: Page Maps

Page Maps

- Partition virtual memory into contiguous ranges called **page**.
  - Every page has the same size.
  - Typically, 4096 bytes.
  - 12 bits to address a byte in a page.
- A virtual address has two components:
  - A page number (20 bits in 32-bit system)
  - An offset into the page (12 bits)

page number (20 bits)	offset (12 bits)

Page Maps (cont.)

- Physical memory is also partitioned into contiguous chucks of bytes called **blocks**.
- A block has the same size as that of a page.
- So, a physical address can be thought of as composing of two components:
  - A block number
  - An offset into bytes of the block

block number (20 bits)	offset (12 bits)

Page Maps (cont.)

- The virtual memory manager maps virtual page numbers to physical page number.
- Address translation process:
  - Translates page number to block number.
  - Concatenating block number with offset.

Page Maps (cont.)

- The mapping is actually implemented as a table called **page table**.
  - Uses a lot of space though.
  - Typically, the map has two levels.
  - Two level page table are used in Intel x86 chips.
- Other implementations are also possible:
  - Linked lists
  - Trees
  - Etc.

Two-Level Page Tables

Figure 5-8. Format of a Linear Address

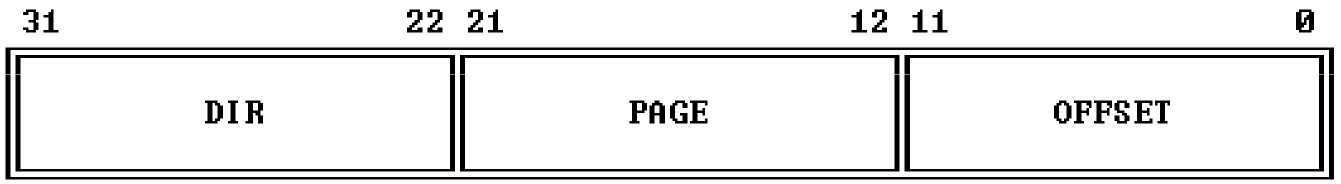

Image Source: Intel 80386 Reference Programmer's Manual

Two-Level Page Tables (cont.)

Figure 5-9. Page Translation

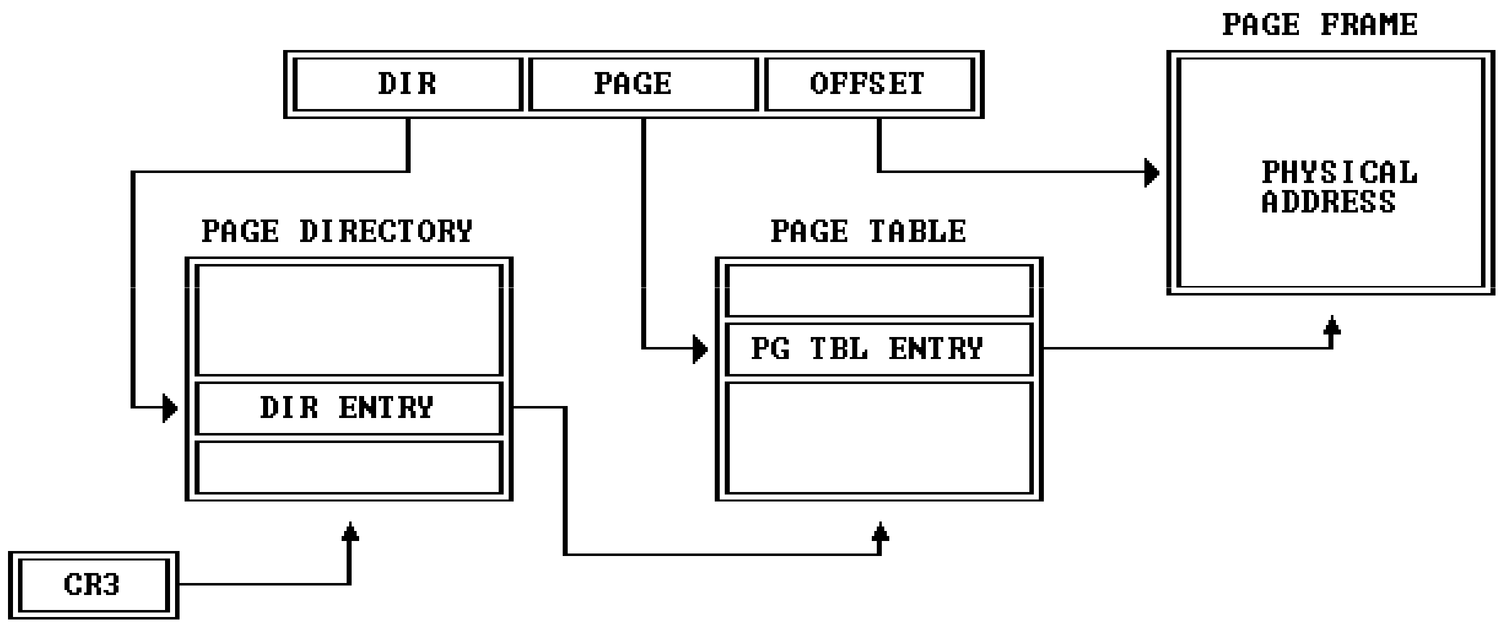

Image Source: Intel 80386 Reference Programmer's Manual

Page Map Implementation

- A page map can be large.
- In most system, it is also dynamic.
  - Software can modify it.
- So, it is kept in physical memory.
  - Usable space is thus reduced.
- Virtual memory manager remembers the physical address of the page map.
  - Most of the time in a special register.

Page Map Format

- Some CPU requires page map to be of a particular format.
  - Intel x86 requires the above two-level page table.
  - Page translation can be done in hardware.
- Some do not.
  - DEC Alpha processor.
  - Translation is done in software.

Typical Page Map Design

- Hardware might specify
  - Page mapping algorithm
  - Page map format.
- Software manages content of page maps.

Caching Page Translation

- Keeping page table in memory means that every memory reference requires actually two memory references.
- This is slow.
- Processor keeps a cache of address translation in on-chip memory.

Translation Look-aside Buffer (TLB)

- An implementation of translation cache.
- Associative memory interface:
  - STORE(virtual-address-key, physical-address-value)
  - READ(virtual-address-key)
- The TLB is very small: 64 - 1024 entries.
- But the key can be any virtual address.
- It is implemented in hardware so is very fast.
- This implementation actually allows software to implement page maps in any possible way.

Address Spaces

Address Spaces

- **Address space** = physical memory a module can read or write.
- Limited those blocks that appear in the module’s page map.
- We can enforce modularity by making sure that the address spaces of modules do not overlap.
- This requires that each module has its own page map.

Supporting Multiple Address Spaces

- The processor has a register called the **page map address register (PMAR)** that holds *physical address* of the current page table being used.
- To transfer control from one module to another, the processor must also change the PMAR to the appropriate value.

Sharing Memory

- Using page maps can also allow controlled sharing of memory between modules.
- How?
  - Two page maps can map to the same block.
- A page map entry can also be supply with additional information:
  - Permission: Can this process performs READ, WRITE, or EXECUTE on this block?
  - Device information: See next page.

Memory-mapped I/O

- A module controls a device by reading/writing memory locations.
- This can be done by mapping a page to a block of a device rather than a memory.
- By mapping a device to some modules’ address spaces and not others’, we can control access to devices.

Address Space Management

Managing Address Spaces

- How do we create/delete address spaces?
- How do we grow address spaces?
- How do we switch one address space to another? Securely?

- Can’t let user modules handle this.
  - Because we don’t trust them not to screw up.

Kernel

- A special module.
- Handles all the above tasks.
- Has its own address space, called the **kernel address space**.
- Kernel space contains *all page tables*.
  - So that the kernel can manage address spaces.

Kernel Memory Management Interface

- CREATE_AS()
  - Create an address space.
- ALLOCATE_BLOCK(block)
  - Allocate a physical block.
- MAP(id, block, page)
  - Map a block at physical address {block} to virtual address {page} in the page table of the module identified by {id}.
- DELETE_PAGE(id, page)
  - Remove mapping of virtual address {page} from the page table of the module identified by {id}.
- DELETE_AS(id)
  - Remove an address space of the module identified by {id}.

What can we do through the kernel?

- Two modules can share blocks.
  - MAP the same block.
- A module can create a new address space its child module.
  - CREATE_AS
  - ALLOCATE_BLOCK
  - MAP the allocated blocks to its own address space.
  - Read program data to the blocks.
  - Map the blocks to the new address space’s page table.

What can we do through the kernel? (cont.)

- Kernel can control:
  - Sharing
  - Protection
  - Device accesses by memory mapping

The catch is . . .

- We need to ensure that everything is done through the kernel!
- We don’t trust user modules.
  - Can’t allow user modules to change the PMAR.
  - Can’t allow user modules to manipulate page tables.
  - Can’t allow user modules to handle interrupts. (If we do so, we allow direct access to devices.)

Enforcing Reliance on the Kernel

- Hardware features:
  - One bit in the process telling that it is running in **kernel mode** or **user mode**.
  - Make it illegal to use instruction that change the PMAR when in user mode.
  - Handle interrupts in kernel mode.
- Simply run user module in user mode, and run kernel in kernel mode.

Switching Address Space

- Since user modules cannot change PMAR, switching address spaces must be done through the kernel.
- Process
  - Switching from address space A to kernel.
  - Kernel writes PMAR with the physical address of the page table of address space B.
- Note that, to switch the module, we have to switch to the kernel first.
- In fact, we reduce the problem to a special case. This approach to solve problems is called **bootstrapping**.

Entering the Kernel

- We require that all modules enter the kernel at one single specified address (of instruction).
- **Gate** = that specified address that served as entry point to another address space.

Entering the Kernel (cont.)

- We actually want modules to jump to the gate if it wants to enter the kernel.
- However, a gate is an address in another address space. The user module cannot see it!

Entering the Kernel (cont.)

- Approaches to solve this problem:
  - Have the kernel shares the block containing the gate with every process.
    - What if the user module jumps somewhere else?
    - Need a hardware mechanism to ensure that doing so is illegal.
  - Have the user module execute a special instruction, the **supervisor call instruction (SVC)**.
    - In x86, this is done by firing a user interrupt.

Entering the Kernel (cont.)

- When the processor enters the gate, it does three things:
  - Change the processor from user mode to kernel mode.
  - Load the PMAR with the address of the kernel page map.
  - Save the program counter (which contains the return address) somewhere, and change the program counter to the gate.

Entering the Kernel (cont.)

- The kernel now has control.
- It can:
  - Check the argument of the supervisor call to see which system call the user module requested.
  - If the transfer is caused by an interrupt, it can also check the interrupt number and branch to the correct interrupt handler.
  - In fact, there can be two or more gates
    - One for SVC.
    - One for interrupts.

  This is done to save the kernel the trouble of distinguishing between the two situations.

Leaving the Kernel

- Once the kernel performed the service the user module asked for, it has to switch to some user module.
- It has to:
  - Load the PMAR to the physical address of the user module’s page table.
  - Reload the program counter that was saved.
  - Change from kernel to user mode.

Things to be Careful About

- The three steps of entering and leaving the kernel must be done as an **atomic operation**.
- **Atomic operation** = it must be done in a single step, without interruption.
- Adverse consequence of not being atomic:
  – If the entering is interrupted after changing to kernel mode, but before loading the PMAR, then a user program might get all access to all the privileged instructions.

Things to be Careful About (cont.)

- Some processors do not do all the three steps of entering/leaving the kernel for you.
- For example, the x86 does not have anything that resemble the leaving kernel mode instruction.
  - In this case, the kernel implementer must deal with all of this by himself.

Things to be Careful About (cont.)

- When you change the PMAR, you change the address space.
- The instruction pointer points to the address in the new address space.
- What’s the next instruction then?
- You really need to be careful about this.

A Toy Implementation

Disclaimer

- This is a toy implementation.
- No system is constructed this way, but very similarly.
- Suspend your disbelief.

Processor

- 32-bit processor.
  - Each register is 32-bit.
  - 32-bit address space.
- PMAR
  - Least significant bit is the user/kernel mode bit.
    - 0 -> kernel
    - 1 -> user
    - This can be done because page table location has to be 4-byte aligned. So the last two bits is not used anyway.
  - Next to least significant bit is interrupt enable bit.
    - 0 -> processor will not check for interrupt
    - 1 -> otherwise
  - When PMAR is 0, there's no address translation.

Processor (cont.)

- SVC
  - Causes the CPU to transfer to a specified location (stored in a register).
  - Has one argument: the identifier of the gate.
  - For example, “SVC 1” might refer to ALLOCATE_BLOCK.
- Privileged instruction can only be executed in kernel mode. This includes setting PMAR.
- Illegal instruction causes the CPU to jump to gates for illegal instruction.

Processor (cont.)

- On entering the kernel (through SVC or interrupt), the processor saves the current instruction pointer on the stack.
- The saved program counter is:
  - Address of illegal instruction in the illegal instruction case.
  - Address of the next instruction to be executed in the interrupt or SVC case.

Booting

- When the system is switched on all registers are zero
  - PMAR is zero. So we start in kernel mode.
  - Instruction pointer is also zero.
- Physical address 0 is the address of the ROM.
- So the system runs the **boot program** burnt to the ROM.

Booting (cont.)

- The boot program loads the kernel from storage from the boot block.
- It stores the kernel in a pre-defined location, say, address KERNEL.
- The boot program then jumps to KERNEL, transferring control to the kernel.

Booting (cont.)

- Kernel then allocates some blocks to use as:
  - Its own stack.
  - Its own page maps.
    - At a predefined address, say KERNELPAGEMAP.
- It fills its own page map.

Booting (cont.)

- Then loads PMAR with KERNELPAGEMAP.
- CAREFUL HERE!
  - Once PMAR is loaded, it will be a whole new address space altogether.
  - How do you ensure that the next instruction is the one you intend it to be?
  - Answer: Always require that the kernel virtual address is the same as physical address.
  - This way, the next instruction is the same whether you load the PMAR or not.

Booting (cont.)

- The kernel then creates the first user process. This process will spawn other user process such as file system service, login service, etc.
- It uses CREATE_AS() to create the address space.
- Then allocate some blocks for the code.
- Where to load the code from?
  - A predetermined location on the disk.
  - This location is built into the the kernel.

Booting (cont.)

- The kernel allocate some more blocks for the first user process.
  - The page table
  - The stack
- To switch the control to the user process, the kernel pushes the address of the first instruction of the user program on the stack.

Leaving the Kernel

- What the kernel have to do?
  - Load the PMAR with the physical address of the user process's page table.
  - Pop the return address from the stack, and jumps to it.

Leaving the Kernel (cont.)

- The code for leaving the kernel is stored at a well-know location, say LEAVING.
- Before using LEAVING, the kernel loads R0 with the physical address of the user page table.

Leaving the Kernel (cont.)

LEAVING:

```
    MOV R0, PMAR //load page table address
    POP R0   //pop return address from stack
    JMP R0   //jump to the return address
```

- Note that the POP instruction pops from the USER PROCESS’S stack because we just changed the PMAR.

Leaving the Kernel (cont.)

- CAREFUL HERE!
  - How can we make sure that the next instruction to execute is the POP after changing PMAR?
- Fill the page table of EVERY user process so that virtual address LEAVING maps to the physical address LEAVING.
- This way, the next instruction is the same whether we change PMAR or not.

Leaving the Kernel (cont.)

- This approach has a problem:
  - What if the user process writes something to virtual address LEAVING?
  - If so, then the whole system may not be able to leave kernel again.
- Two approaches to deal with this:
  - Copy the code to every process's address space.
  - Set permission on the page with address LEAVING so that user processes can only READ and EXECUTE from it.
  - The second approach is less wasteful and more successful.

Entering the Kernel

- Similar to leaving, we put entering code at address ENTERING.
- Inform the CPU that ENTERING is a gate.
- What do we have to do to enter the kernel?
  - The CPU has already changed the mode for us.
  - It also has saved the return address on the stack of the user process for us.
  - So all we have to do is changing the PMAR.

Entering the Kernel (cont.)

```
ENTERING:
    MOV KERNELPAGEMAP, PMAR
    JMP {somewhere}
```

- Again, virtual address ENTERING must map to the same physical address for all process to ensure that the next instruction is the one intened.

Taking Things a Step Further

- Every user process address space has to have two areas that it cannot write to, and those areas map to the same physical address.
  - LEAVING
  - ENTERING
- Some systems take another step:
  - Cut a portion of the user process address space, and maps the kernel address space to that.
  - In this way, there's no need to load PMAR when entering the kernel.
  - The kernel can also modify the data in user process's address space very easily.
    - Just use the normal LOAD and STORE.