CS-446/646

Virtual Memory

C. Papachristos
Robotic Workers (RoboWork) Lab
University of Nevada, Reno

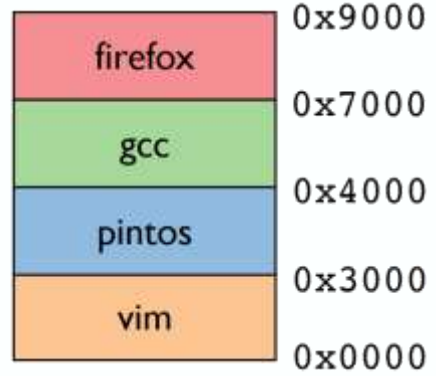
Virtual Memory (VM)

Early computers

- Programs use *Physical Memory* addresses directly
- OS loads job, runs it, unloads it

- *Multiprogramming* paradigm changed everything
  - Want to have multiple *Processes* in Memory simultaneously

- Consider *Multiprogramming* directly on *Physical Memory*
  - What happens if OS (Pintos) needs to expand?
  - What if **`vim`** needs more *Memory* than is on the Machine?
  - What if OS (Pintos) tries to erroneously write to *Memory Address* **`0x7100`**?
  - When does **`gcc`** have to know it will run at **`0x4000`**?
  - What if **`vim`** isn't using most of its *Memory*?



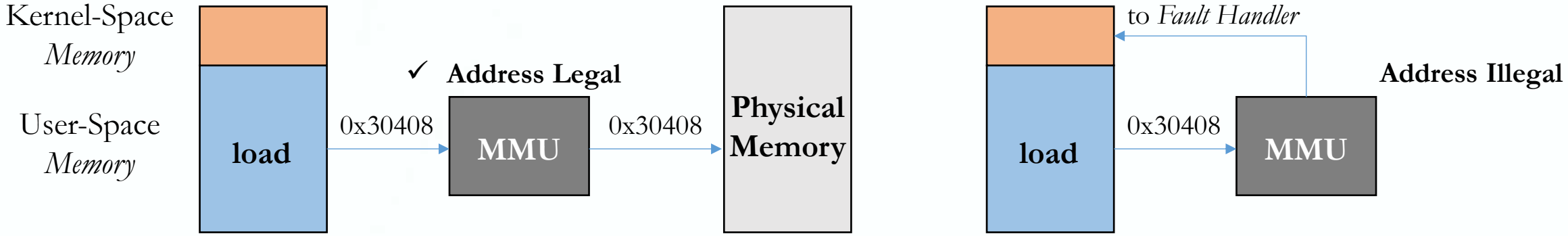
Virtual Memory (VM)

Issues of sharing *Physical Memory*

- Protection
  - A bug in one *Process* can corrupt *Memory* in another
  - Must somehow prevent *Process* A from clobbering *Process* B's *Memory*
  - Also prevent *Process* A from even being able to observing *Process* B's *Memory*
- Transparency
  - A *Process* shouldn't require specific *Physical Memory* locations
  - *Processes* often require large amounts of *Contiguous Memory* (Program *Stack*, large Data Structures, etc.)
- Resource exhaustion
  - Developer typically assumes Machine has "enough" *Memory*
  - Sum of sizes of all *Processes* often greater than *Physical Memory*

Virtual Memory

Virtual Memory Goals



- Give each Program its own *Virtual Address Space*
  - At runtime, *Memory Management Unit* (MMU) translates each **`load`**/**`store`**
  - Application doesn't see *Physical Memory* Addresses
- Enforce protection
  - Prevent one *Process* from messing with another's *Memory*

Allow Programs to "see" more *Memory* than exists

- Ability to relocate some *Memory* sections and their accesses to Disk

Virtual Memory Advantages

Can re-locate *Program* while running

- Run partially in *Memory*, partially on Disk

Most of a *Process*'s *Memory* may be idle (*"80/20 Rule"* – more later)

- Can write idle parts to Disk until they become needed
- Let other *Processes* use *Memory* of idle part
  - Like CPU *Virtualization*: When a *Process* is not using CPU, switch-over to executing another *Process*
    When a *Memory* region is unused, "switch" it to another *Process*

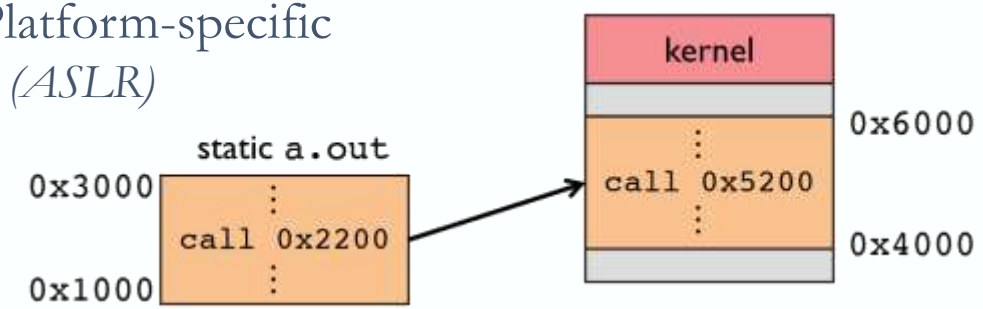
Challenge:

- *Virtual Memory* introduces an additional layer to *Memory* accessing
  - Can impact Performance

Core Concept 1: Load-Time *Dynamic Linking*

- Compiler makes distinguishable in the *Symbol Table*:
  - *Global Definitions* which are routines/variables exported by the file being Compiled
  - *External Refs* corresponding to external routines/variables
- Linker has to "patch" at **Loading-Time** the Addresses of Symbols

  i.e. Load-Time *Dynamic Linking* happens when *Process* executed (not at Compile-Time or at Link-Time)
  - First determine where *Process* will reside in *Memory*; Platform-specific
    - e.g. techniques like *Address Space Layout Randomization (ASLR)* which guards against *Buffer Overflow* attacks
  - Then adjust all references within Program (adding)



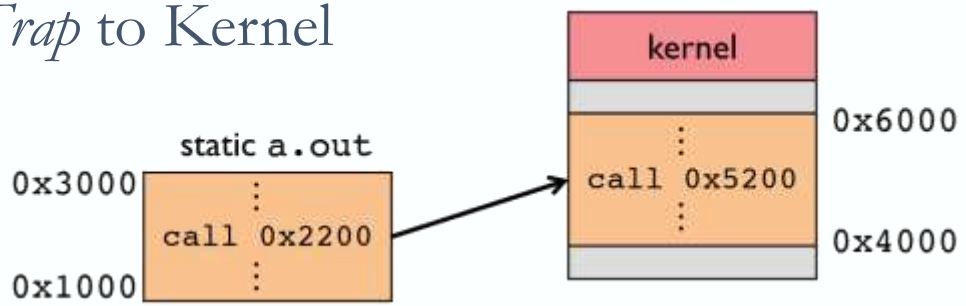
Problems?

- "Patching" is required for each run, and is time-consuming
- How to move an entire *Process* that is already in *Memory*?
- What if no *Contiguous* free region fits can fit the Program?

Core Concept 2: *Base & Bound* Register

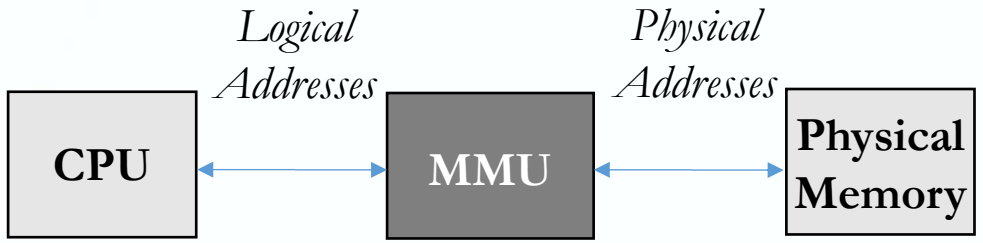
Two special *Privileged Registers*

- *Base* and *Bound*
- On each **`load`**/**`store`**/**`jump`**:
  - *Physical Address* = *Base* + *Virtual Address*
  - Check (0 ≤ *Virtual Address* < *Bound*), else *Trap* to Kernel
- So to move *Process* in *Memory*:
  - Change *Base Register*
- What happens on *Context Switch*?
  - OS has to reload *Base* and *Bound Registers*



Core Concept 3

- Programs **`load`**/**`store`**/**`jump`** to *Virtual Addresses*
- Actual *Memory* uses *Physical Addresses*



The *Virtual-to-Physical Memory*-managing Hardware is the *Memory Management Unit (MMU)*

- Usually part of CPU
  - Can be configured through *Privileged Instructions* (e.g. Loading of *Bound Reg*)
- Translates from *Virtual*-to-*Physical* Addresses
- Gives a per-*Process* view of the *Memory*, called the "*(Virtual) Address Space*"

Base & *Bound* Trade-offs

Advantages

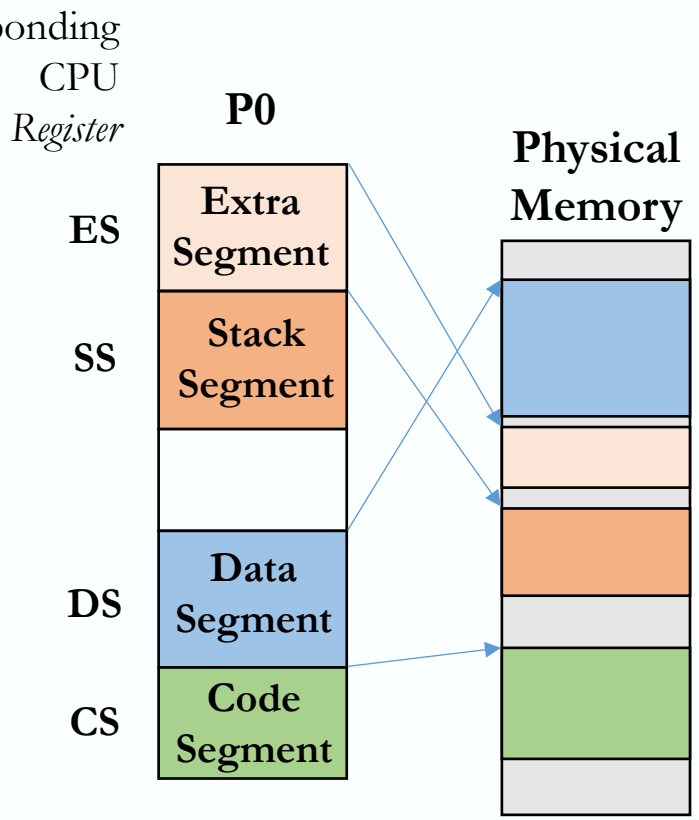
- Cheap in terms of Hardware (just 2 *Registers*)
- Cheap in terms of cycles
  - Perform adding *Logical Address* and *Base* (to determine *Virtual-to-Physical* mapping), and compare (to ensure *Bound* within limits) **in parallel** inside the MMU

Disadvantages

- If we rely on using just this extra layer:
  - Growing a *Process' Address Space* is expensive or impossible
  - No way to share code or data

Core Concept 4: *Segmentation*

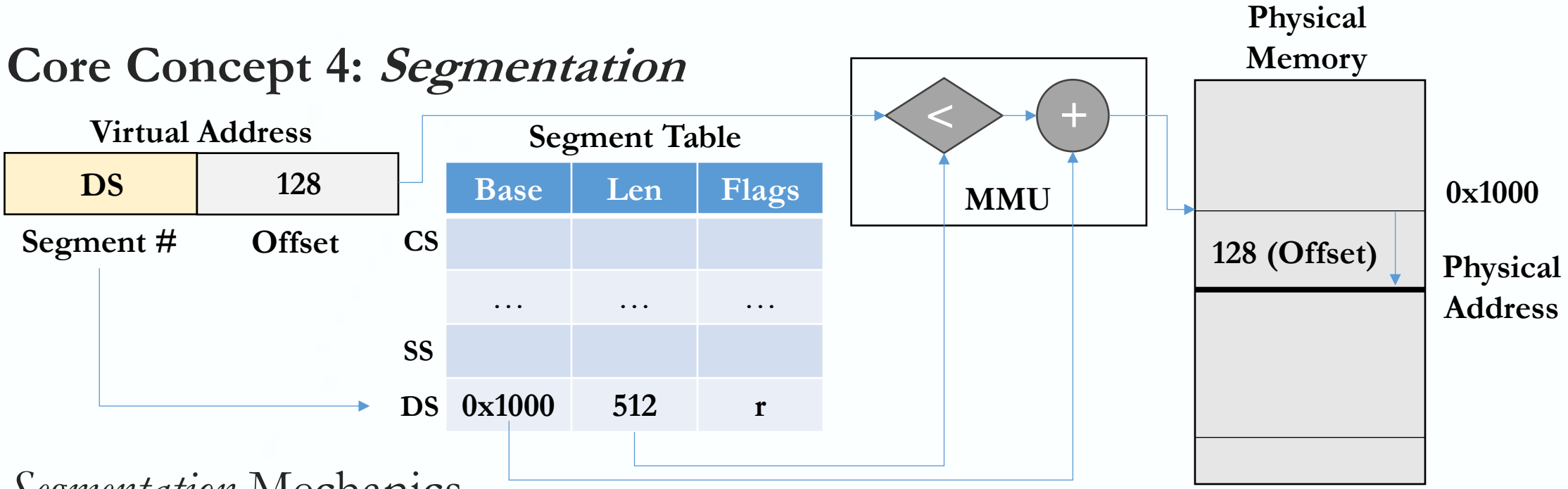
- Let *Processes* have multiple *Base* & *Bound Regs*
  - *Address Space* built from many *Segments*
  - Can share/protect *Memory* at *Segment*–level granularity
- The *Virtual Address* has to somehow indicate the specific *Segment* that it corresponds to
  - Has to contain this as part of its information



Example Illustration:
8086 *Segmentation* Model

Virtual Memory

Core Concept 4: *Segmentation*



Segmentation Mechanics

- **Each** *Process* has its own *Segment Table*, set up by the OS
- Each *Virtual Address* indicates a *Segment* and *Offset*
  - Top bits of *Virtual Address* select *Segment*, low bits select *Offset*
  - x86 stores each *Segment*'s *Physical Address* in *Registers* (**CS**, **DS**, **SS**, **ES**, **FS**, **GS**)
    - *Instruction* Fetch uses **CS**; *Memory* Addressing defaults to **DS** / **SS**; Some (string) *Instructions* use **ES**, **FS** & **GS** are special-purpose (e.g. Linux uses **GS** for *Thread-local* storage)

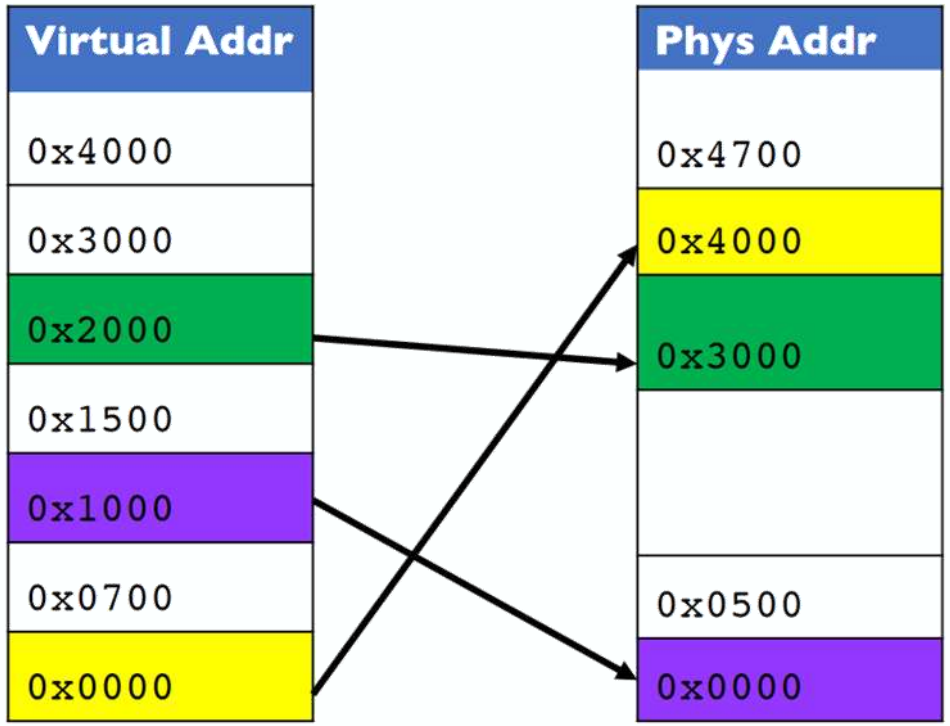
Core Concept 4: *Segmentation*

Segmentation Example

A *Process' Segment Table*

Segment	Base	Bound	RW
CS	0x4000	0x6ff	10
SS	0x0000	0x4ff	11
DS	0x3000	0xfff	11
...			00

- 4-bit *Segment* # (1st digit)
- 12-bit *Offset* (last 3 digits)



Segmentation Trade-offs

Advantages

- Multiple *Segments* per *Process*
- Can even facilitate *Shared Memory* (OS sets up each *Process*' *Segment* Mappings)
- Don't need entire *Process* to reside in *Memory* at all times
- Variable *Segment Boundaries*

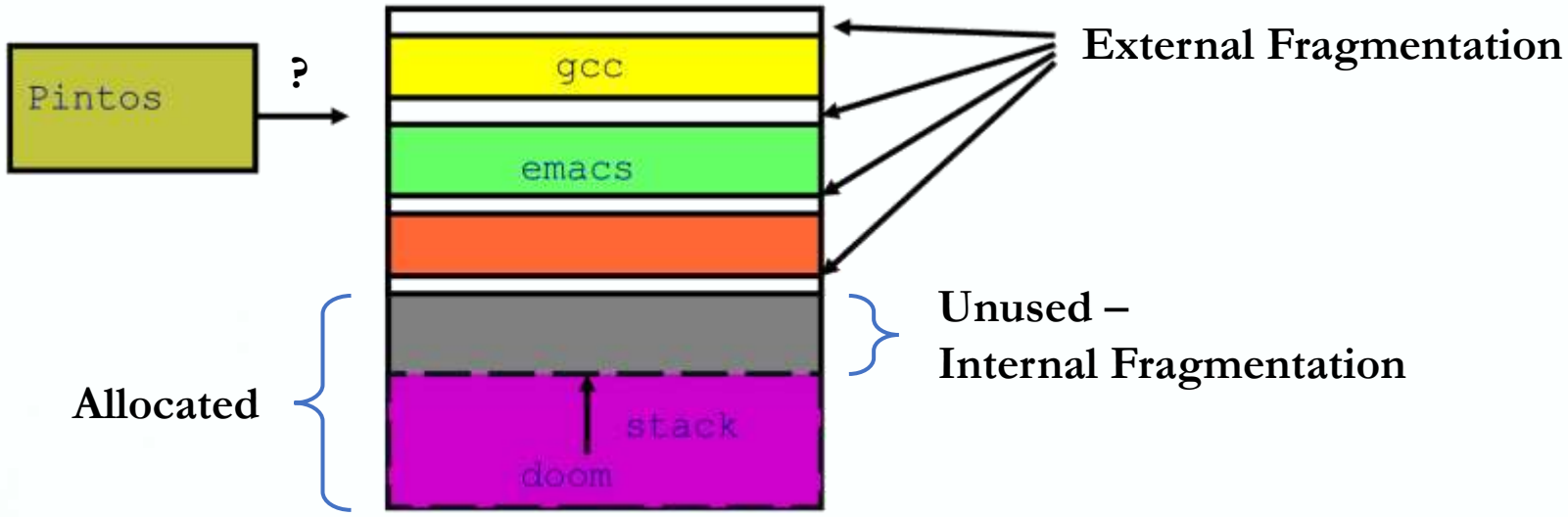
Disadvantages

- Requires MMU Translation Hardware, which can impact performance
- *Segments* can overlap (e.g. many *Segment-Offset* combinations that can map to same *Address*)
- *Segment* mechanics not completely transparent to Program
  - e.g. *Default Segment* faster or uses shorter *Instructions*
    Overriding use of the *Default Segment* possible with special *Instruction* override bytes ("*Far Call*")
- A ***n***-byte *Segment* needs ***n*** *Contiguous* Bytes of *Physical Memory*
  - *"Fragmentation"* becomes real problem

Fragmentation

Fragmentation → Inability to use Free *Memory*

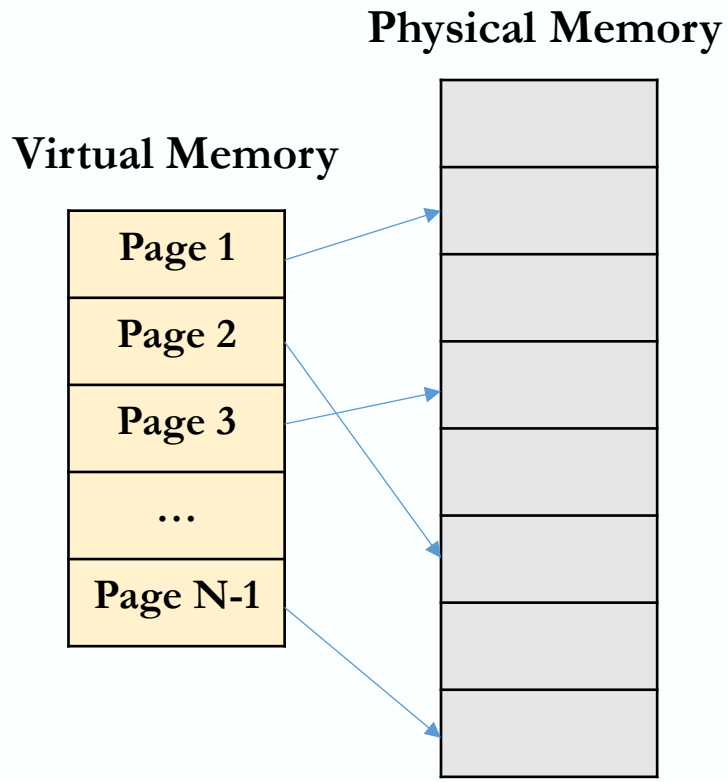
- Over time, have to suffer from either of:
  - *External Fragmentation*: Variable-sized pieces ≡ Many small holes
  - *Internal Fragmentation*: Fixed-sized pieces ≡ No external holes, but forced internal wasting



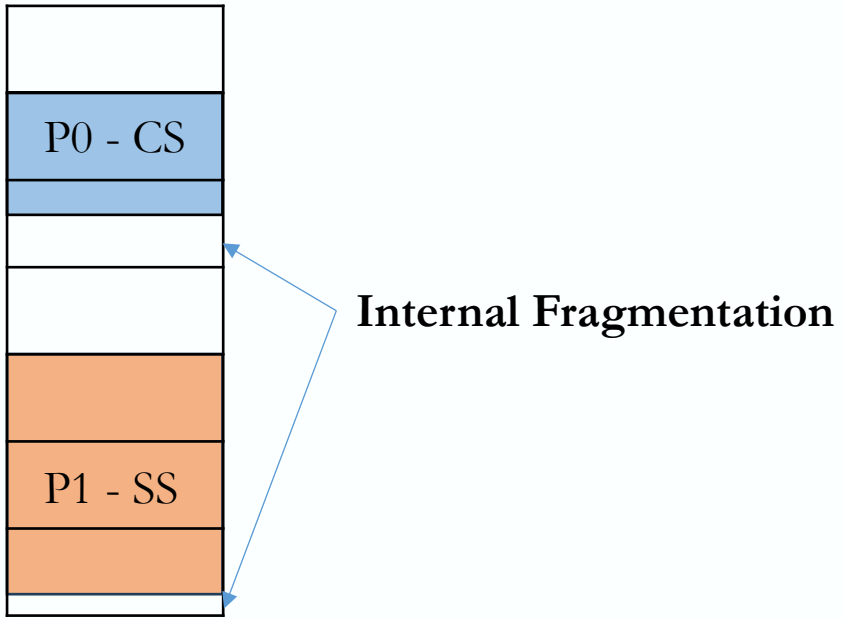
Virtual Memory

Core Concept 5: *Paging*

- Divide *Memory* up into **fixed-size** *Pages*
  - Eliminates *External Fragmentation*
- Mapping of *Virtual Pages* –to– *Physical Pages*
  - A higher level of *Virtual Memory* Mapping
  - **Each** *Process* has a separate such Mapping
- Allow OS to gain control on certain operations
  - Read-only *Pages* trigger *Trap* to OS on-*Write*
  - Invalid *Pages* trigger *Trap* to OS on-*Read* or on-*Write*
  - OS can change Mapping and resume Application

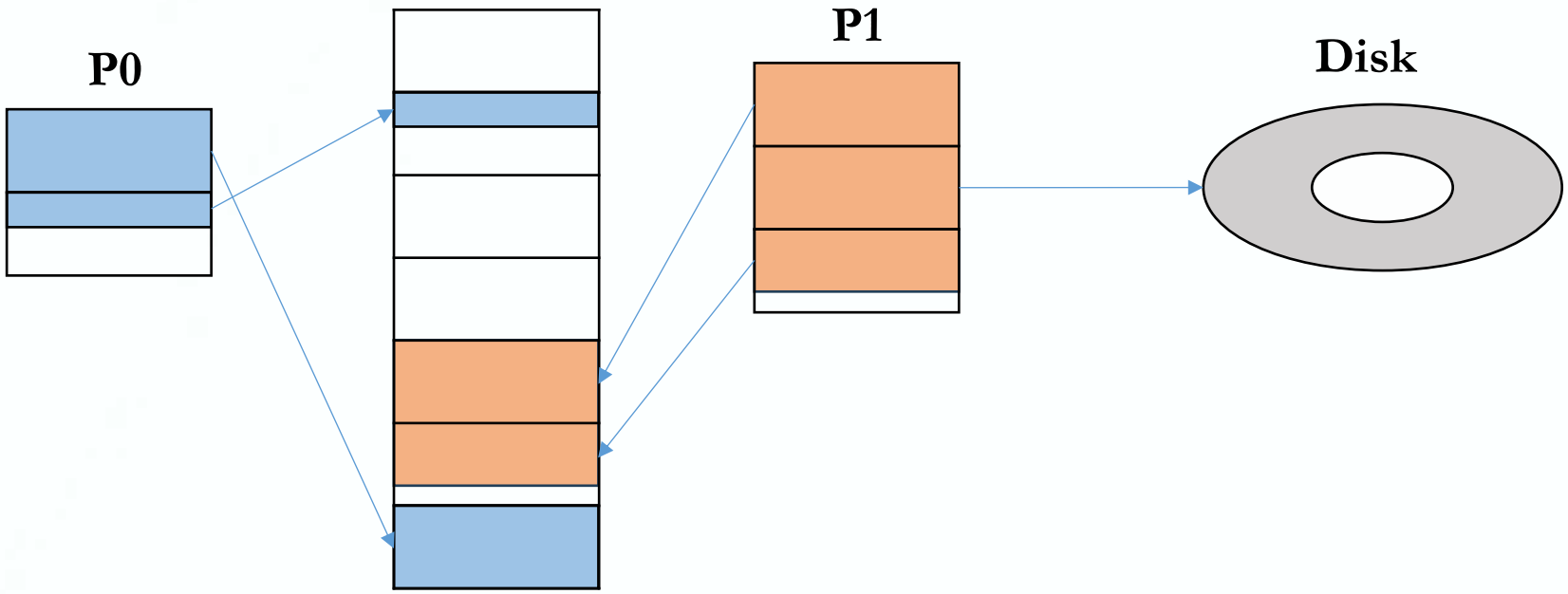


Paging Trade-offs



- Eliminates *External Fragmentation*
- Simplifies Allocation, Freeing, and *Memory* Backing with storage ("*Swap*")
- Average *Internal Fragmentation* of .5 *Pages* per *Segment*

Simplified Allocation



- Allocate any *Physical Page* to any *Process*
- Can store idle *Virtual Pages* on Disk ("Paging-Out")

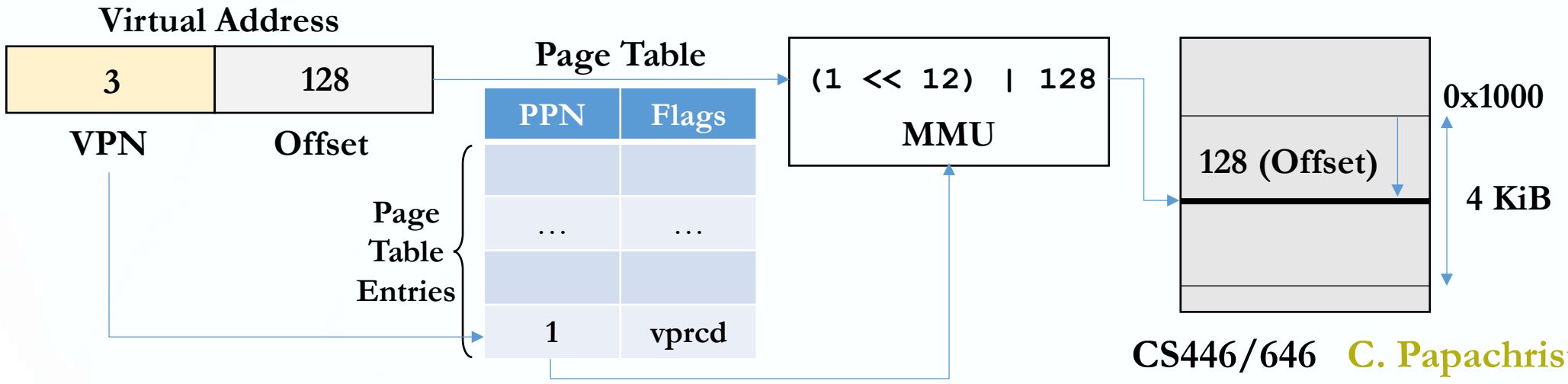
Paging Data Structures

- *Pages* are fixed-size, e.g. 4KiB on a 32-bit system
  - *Virtual Address* has 2 parts: *Virtual Page Number* and *Offset*
  - Least Significant 12 Bits ($2^{x=?}$=4 KiB or $x=log_2$(4 KiB)=12) of Address are *Page Offset*
  - Most Significant Bits are *Page Number*
- *Page Tables*

  For **each** Process the OS keeps a *Page Table*
  - Map *Virtual Page Number* (VPN) to *Physical Page Number* (PPN)
  - VPN is the index into the *Page Table* that determines PPN
  - PPN also called *Page Frame Number*
  - *Page Table* contains one *Page Table Entry* (PTE) per each *Page* of the *Address Space*
  - *Page Table Entry* also includes bits for *Protection*, *Validity*, etc. (more later)

Paging Data Structures

- *Pages* are fixed-size, e.g. 4KiB on a 32-bit system
  - If the *Virtual Address* is 32 bits (2^{32} *Physical Addresses*), and the system uses 4 KiB *Pages* (2^{12} *Offsets* per PTE) then a *Process' Page Table* can have 2^{20} (1 million) *Page Table Entries* max (with each PTE being 4 Bytes in size)
- On *Memory* access: Translate VPN to PPN, then add *Offset*



Page Table Entries

- *Page Table Entries* control Mapping

Physical Page Number	V	P	R	C	D

  - The *Physical Page Number* (*PPN*) determines the *Physical Page*
  - The *Valid* (or *Present/Absent*) bit indicates whether the *Page* exists in *Memory*
    - Checked each time the *Virtual Address* is tries to use this PTE
    - *Page Fault* when *Absent*
  - The *Protection* bits say what operations are allowed on this *Page*
    - Read, Write, Execute
    - *Protection Fault* on not allowed operation
  - The *Reference* (or *Access*) bit says whether the *Page* has been accessed
    - It is set when a read or write to the *Page* occurs
  - The *Modified* (or *Dirty*) bit says whether the *Page* has been written to
    - It is set when a write to the *Page* occurs
  - The *Caching* bit enables/disables *Caching* of the *Page*
    - Related to *Translation Lookaside Buffer (TLB)* – more later

Paging Advantages

- Easy to Allocate *Memory*
  - *Memory* comes from a "*Freelist*" of fixed-size chunks
  - Allocating a *Page* is just removing it from the "*Freelist*" (more in *Dynamic Allocation*)
  - *External Fragmentation* not a problem
- Easy to swap out chunks of a Program
  - All chunks are the same size
  - Use *Valid* bit to detect references to "*Swapped*" *Pages*
  - *Pages* are a convenient multiple of the Disk *Block Size* (more in *Disks* & *Filesystems*)

Paging Limitations

- Does not alleviate *Internal Fragmentation*
  - Process may not use *Memory* in whole multiples of a *Page*
- *Memory* referencing overhead
  - At least 2 references per Address Lookup (*Page Table* first, then *Memory Lookup*)
    - x86 requires *Page Directory*, *Page Table*, then *Memory Lookups*
  - Solution: Use a Hardware *Cache of Lookups* – the TLB (more later)
- Memory required to hold contents of *Page Table* can be significant
  - Need one PTE per *Page*
  - 32-bit *Address Space* covered by 4 KiB *Pages* → Need 2^{20} PTEs (2^{32} Tot. Addresses / 2^{12} Addresses per-*Page*)
  - 4 Bytes / PTE = 4 MiB for a *Page Table*
  - 25 *Processes* = 100 MiB just for the *Page Tables*
  - Solution: *Multi-Level Page Tables* (more later)

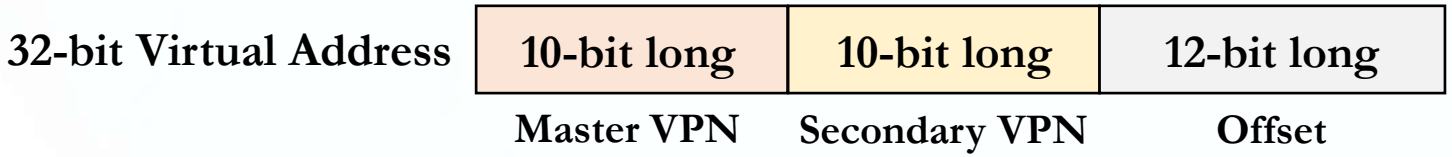
Managing *Page Tables*

- Why use 4 KB *Pages* (e.g. instead of 8 KB or even 4 MB – Linux "Huge pages")
  - Empirical choice, but not smaller because:
    - More *Page Tables* needed,
    - Likely more *Page Faults*
  - Also, larger *Page Tables* incur increased *Internal Fragmentation*
- Size of the *Page Table* for a 32-bit *Address Space* w/ 4 KiB *Pages* (12-bit *Offset* field)
  - $(2^{32} / 2^{12})$ PTEs $\times$ 4 B/PTE $=$ 4 MiB
  - Too much overhead to incur for every single *Process* spawned
- Observation: We really only need to map the *Address Space* portion actually being used
- How to map only what is actually being used?
  - Could dynamically extend *Page Table*…
    - Does not work if *Address Space* is sparse (*Internal Fragmentation*)
  - Use one more level of indirection: *Two-Level Page Tables*

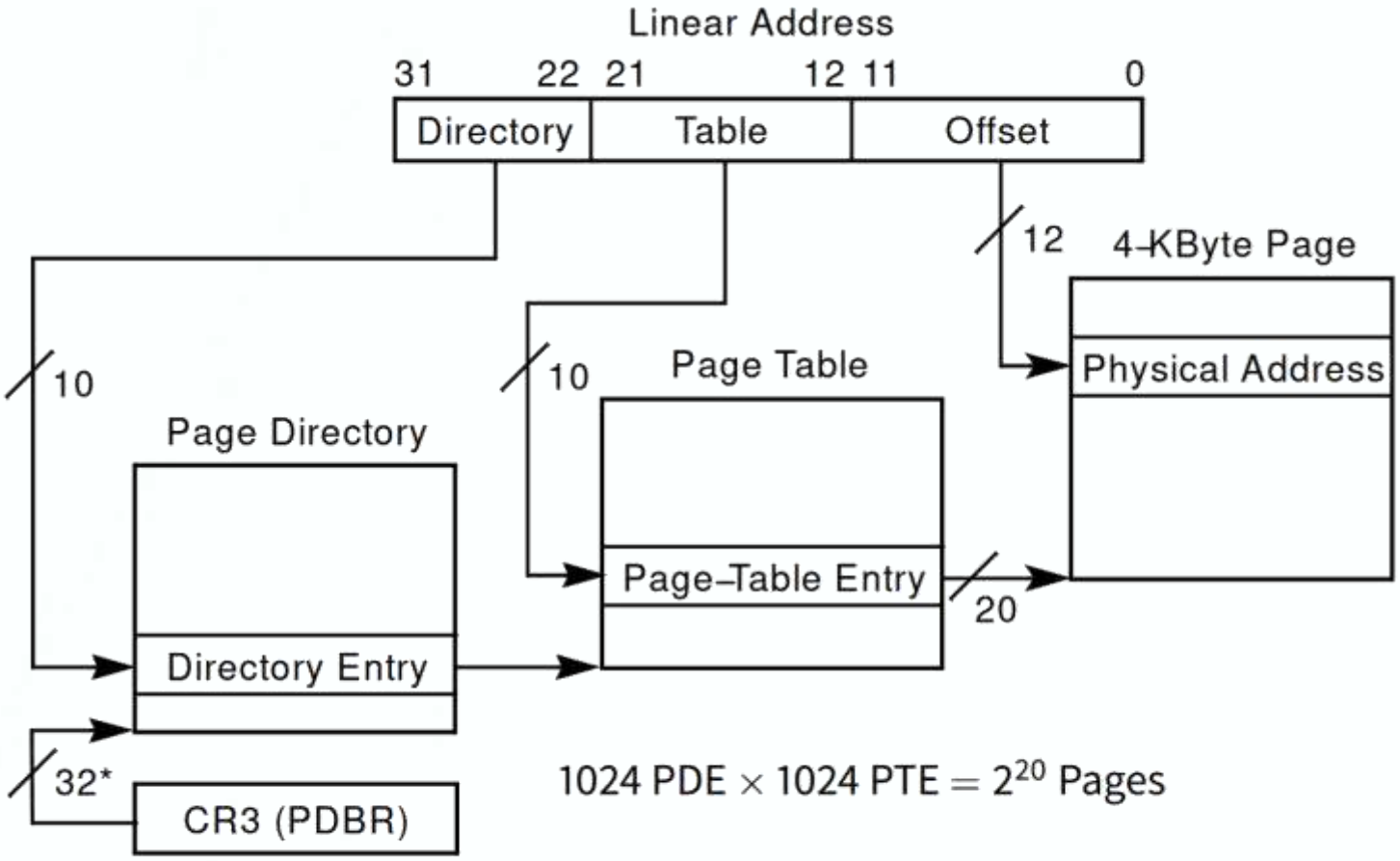
Two-Level Page Tables

Virtual addresses (VA)s have 3 parts:

- *Page Directory Number (/Master Page Number)*
  - **One** *Page Directory (/Master Page Table)* per-*Process*, that maps VAs to one of the *Secondary Page Tables*
- *(Secondary) Page Number*
  - **Multiple** *(Secondary) Page Tables* per-*Process*, each mapping *Secondary Page Numbers* to *Physical Pages*
- *Offset*
  - Where in the *Physical Page* the specified Address is located
- Example
  - a) Page Size: 4 KiB - b) PTE Size: 4 B (32-bit) - c) Offset Size: $log_2(4\ \text{KiB}) = 12\text{-bit}$
  - Want to fit entire *Master Page Table* in 1 *Page*: 4 KiB / 4B = 1 Ki = 1024 entries (*Secondary Page Tables*)
  - For 1024 entries, $2^{x=?}=1\ \text{Ki}$ or $x=log_2(1\ \text{Ki}) = 10$ bits for *Master VPN* → Remaining 10 bits for *Secondary VPN*



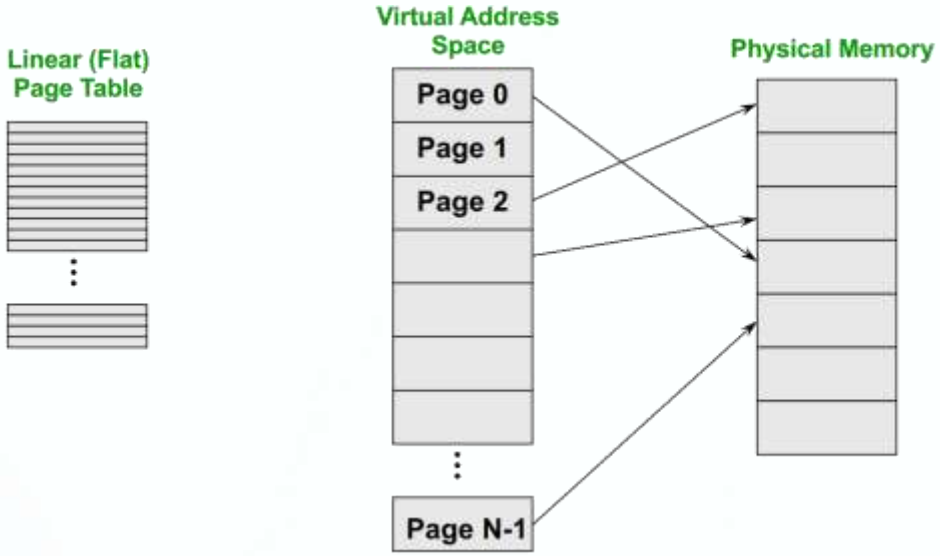
x86 *Page Translation*



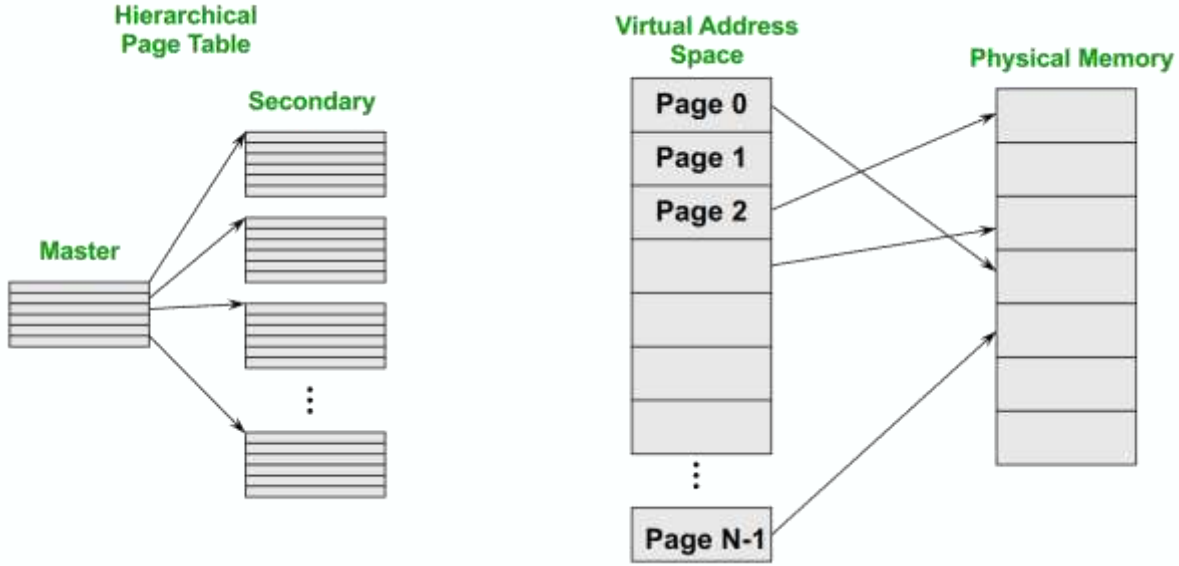
Two-Level Page Tables

Evolution

Page Tables



Two-Level Page Tables



Two-Level Page Tables

Evolution (continued)

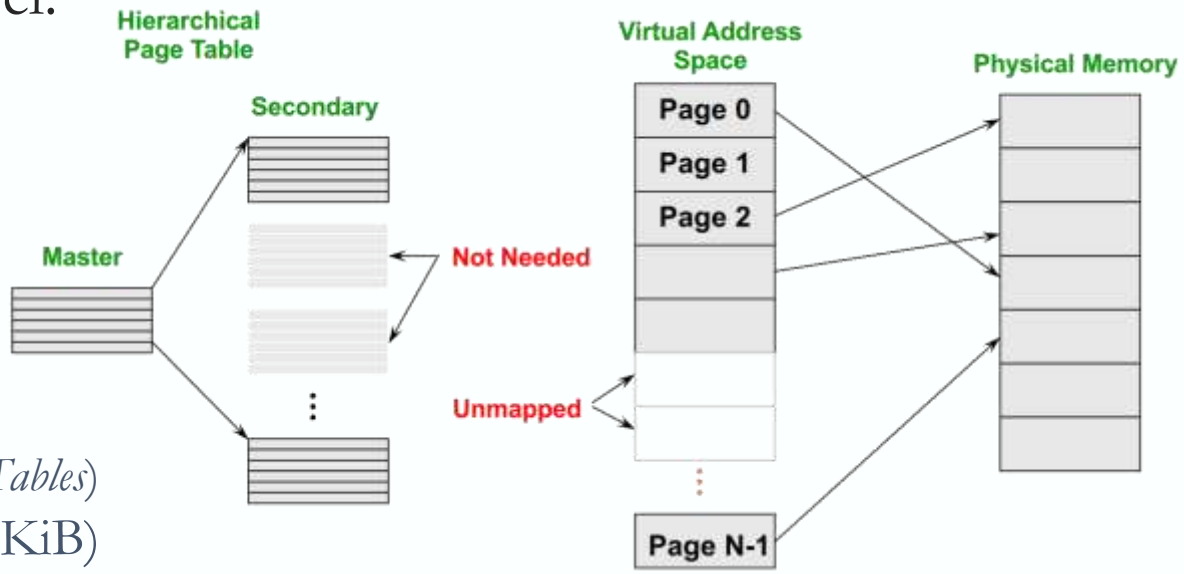
Two-level Page Tables aim to reduce the overhead of storing *Page Tables*, but with just adding 1 more level:

- Each *Page Table* previously costed 4 MiB to store ($(2^{32} / 2^{12})$ PTEs × 4 B/PTE)

Now, with 1024 *Secondary Page Tables*

- Each *Secondary Page Table* has 2^{10} PTEs, thus has a size of 4 KiB
- Size of all *Secondary Page Tables* is 1024 × 4 KiB = 4 MiB (as with single-level *Page Tables*)
- + we also need to store the *Master Page Table* (4 KiB)

Two-Level Page Tables



- But we only need to allocate *Secondary Page Tables* which correspond to *Master Page Table* entries that hold Virtual Address **ranges** that are *Valid*

x86 *Paging*

- *Paging* enabled by bits in a *Control Register* (**`%CR0`**)
  - Only *Privileged* OS code can manipulate *Control Registers*
- Normally 4 KiB *Pages* (Architecture may support other sizes)
  - **`%CR3`**: Points to *Physical Address* of 4 KiB *Page Directory* (/*Master Page Table*)
  - e.g. **`pagedir_activate()`** in Pintos (**`userprog/pagedir.c`**)
- *Page Directory*: 1024 *Page Directory Entries* (PDE)s
  - Each contains *Physical Address* of a *Page Table*
- *Page Table*: 1024 *Page Table Entries* (PTE)s
  - Each *Page Table Entry* contains *Physical Address* of a *Virtual* 4 KiB *Page*
  - Each *Page Table* covers 4 MiB of *Virtual Memory* (1024 × 4 KiB)

x86 *Paging* and *Segmentation*

- The IA-32 (x86-32 bit) Architecture supports both *Paging* and *Segmentation*
  - *Segment Register Base* + *Pointer Val* = Linear Address
  - *Page Translation* happens on Linear Addresses
- Two levels of Protection and Translation checking
  - *Segmentation* model: Architecture supports four *Privilege Levels* (CPL 0-3)
  - *Paging* model: Architecture supports only two, so 0-2 = *Kernel-level*, 3 = *User-level*
- Why do we want both *Paging* and *Segmentation* ?

x86 *Paging* and *Segmentation*

- Why do we want both *Paging* and *Segmentation* ?
- Short answer: We don't – just adds overhead
  - Most OSes simulate "Flat Memory Model"
  - IA-32 (x84-32) cannot turn off *Segmentation*, so OS sets *Base* = 0, *Bounds* = 0xFFFFFFFF in all *Segment Registers*, then forget about it (i.e. does not actively leverage *Segmentation* support)
  - x86-64 Architecture removes much *Segmentation* support
- Long answer: Has some fringe/incidental uses
  - Use:
    - *Segments* for logically related units
    - *Pages* to partition *Segments* into fixed-size chunks
      - Tends to be complex
  - VMware runs guest OS in *Privilege Level* CPL 1 to *Trap Stack Faults*

Where does the OS live?

- Where does the OS live?
  - In its own separate *Address Space*?
    - Can't do this on most Hardware (e.g. **`syscall`** *Instruction* won't switch *Address Spaces*)
    - Also would make it harder to parse any **`syscall`** arguments passed as Pointers
- So, it "shares" the same *Address Space* as the *Process*
  - I.e. there exist some "shared" (same) *Page Table* Mappings
    - *Remember*: Each *Process* has its own separate *Page Table* Mappings
  - Can use *Protection* bits to prohibit *User* code from accessing *Memory* belonging to the Kernel-owned *Address Space*

Note: Recent *Spectre* and *Meltdown* CPU vulnerabilities force OSes to reconsider things

- https://lwn.net/Articles/743265/

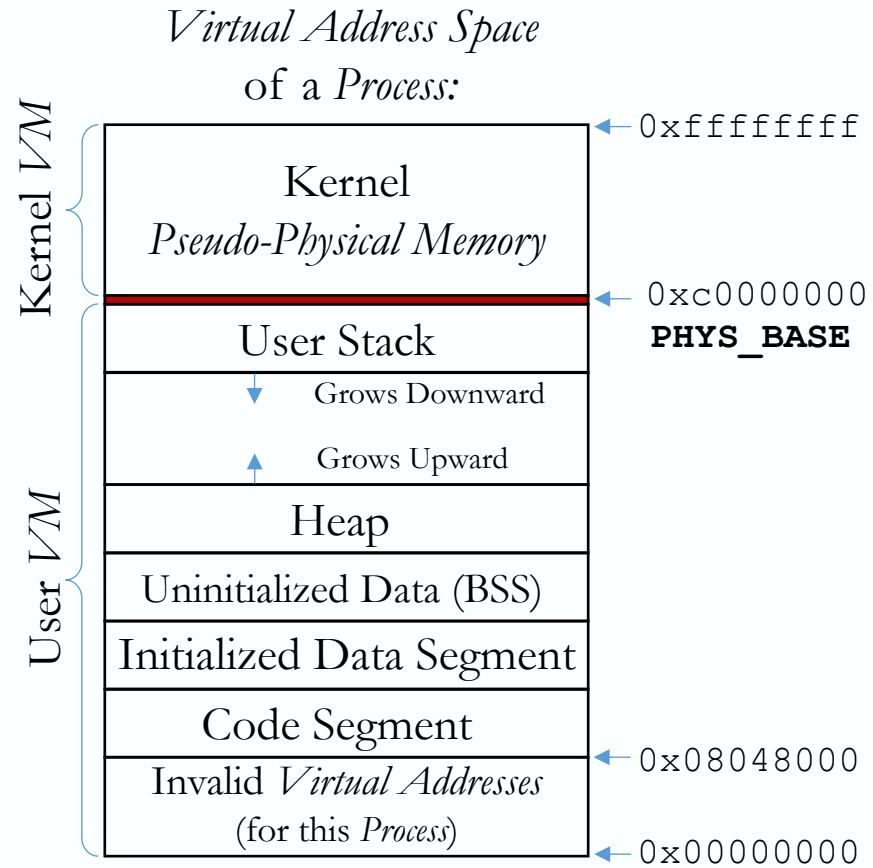
Virtual Memory

Where does the OS live?

- Typically all Kernel *Text*, most *Data* are at same *Virtual Address* **for every** *Address Space*
- I.e. for every per-*Process Page Table* Mapping
  - On x86, must manually set up *Page Tables* for this
  - Usually just map Kernel into contiguous *Virtual Memory* at the phase when Bootloader puts Kernel into contiguous *Physical Memory*
    - e.g. in Pintos, Kernel *Virtual Memory* is mapped one-to-one to *Physical Memory*, starting at **`PHYS_BASE`**, and a User Program can only access its own User *Virtual Memory*. An attempt to access kernel *Virtual Memory* causes a *Page Fault*, handled by **`page_fault()`**.

    "Identity Mapping"
  - In some cases Hardware can also place *Physical Memory* (Kernel-only) somewhere in *Virtual Address Space*

Pintos *Virtual Memory* model



Virtual Memory

Where does the OS live?

- The Kernel *Virtual Memory* Mappings are the same across all *Processes*

Implications:

- When we *Context Switch* to another *Process*, although it involves changing the *Page Tables*, the Kernel *Virtual Memory* Addresses are still valid after the switch
- All objects created in the Kernel functions are accessible across *Processes*
  - e.g. `static struct list all_list;` `threadX->wait_status;`
  - For example, allows to implement `int wait (pid_t pid)` where you want to create a variable in `struct thread` to store some information of the new *Child Process* (`wait_status`), and it is necessary to be able to read/write this variable from the *Parent Process*
- *Memory* for User *Processes* will be `free()`d when it exits, but *Memory* objects allocated within the Kernel code using `malloc()` should be explicitly `free()`d

Where does the OS live?

- The Kernel *Virtual Memory* mappings are the same across all *Processes*

How ?

- e.g. Pintos

```
struct thread {
  tid_t tid; // Thread identifier
  enum thread_status status; // Thread state
  char name[16]; // Name (for debugging
purposes)
  uint8_t *stack; // Saved stack pointer
  int priority; // Priority
  struct list_elem allelem; // List element for
                            // all threads list
  struct list_elem elem; // List element

  #ifdef USERPROG
  /* Owned by userprog/process.c. */
  uint32_t *pagedir; // Page directory
  #endif

  /* Owned by thread.c. */
  unsigned magic; // Detects stack overflow
};
```

```
bool load (const char *file_name, void (**eip) (void), void **esp) {
  struct thread *t = thread_current ();
  ...
  /* Allocate and activate page directory. */
  t->pagedir = pagedir_create ();
  if (t->pagedir == NULL)
    goto done;
  process_activate ();
  /* Open executable file. */
  file = filesys_open (file_name);
  ...
}
```

```
uint32_t * pagedir_create (void) {
  uint32_t *pd = palloc_get_page (0);
  if (pd != NULL)
    memcpy (pd, init_page_dir, PGSIZE);
  return pd;
}
```

`init_page_dir`: Initialized in **`paging_init()`** in **`thread.c`**

Where does the OS live?

- The Kernel *Virtual Memory* mappings are the same across all *Processes*

How ?

- e.g. Pintos

```
bool load (const char *file_name,
           void (**eip) (void),
           void **esp) {
  struct thread *t = thread_current ();
  ...
  /* Allocate and activate page directory. */
  t->pagedir = pagedir_create ();
  if (t->pagedir == NULL)
    goto done;
  process_activate ();
  /* Open executable file. */
  file = filesys_open (file_name);
  ...
}
```

```
void process_activate (void) {
  struct thread *t = thread_current ();
  pagedir_activate (t->pagedir);
  /* Set thread's kernel stack for use in processing interrupts. */
  tss_update ();
}
```

After this point *Virtual Memory* mappings have changed

```
void pagedir_activate (uint32_t *pd) {
  if (pd == NULL)
    pd = init_page_dir;
  /* Store the physical address of the page directory
     into CR3 aka PDBR (Page Directory Base Register).
     This activates our new page tables immediately.
     See [IA32-v2a] "MOV—Move to/from Control Registers"
     and [IA32-v3a] 3.7.5 "Base Address of the Page Directory". */
  asm volatile ("movl %0, %%cr3" : : "r" (vtop (pd)) : "memory");
}
```


Addressing *Page Tables*

Where do we store the *(Secondary) Page Tables* (which *Address Space*)?

- Possibility #1: *Physical Memory*
  - Easy to address, no Translation required
  - But, allocated *(Secondary) Page Tables* consume *Memory* for lifetime of each *Virtual Address Space*
- Possibility #2: *Virtual Memory* (OS *Virtual Address Space*)
  - "Cold" (unused, more on that later) *Pages* can be "*Paged-Out*" to Disk as we've seen so far
  - So, entire *(Secondary) Page Tables* (each being the same size as a single *Page*) can also be "*Paged Out*"
    - But! We keep only the *Page Directory (/Master Page Table)* from being "*Paged-Out*"
      - called "*Wiring*"

If we're "*Paging-Out*" *(Secondary) Page Tables*, could as well *Page-Out* the entire OS *Address Space*

- Some special code and data (*Fault*, *Interrupt* Handlers) need to remain "*Wired*"

Addressing *Page Tables*

- Kernel *Address Space* is also Mapped
  - First, *Identity Mapping*:
    - i.e. *Virtual Addresses* and *Physical Addresses* are the same
    - Generally up to 4 MiB
  - After *Identity Mapping* the Kernel is Mapped to some other *Virtual Addresses*
    - e.g. Linux, Windows
- A *Process*' PCB contains the *Physical Address* (remember: *"Wired"*) of its *Page Directory*
  - This *Physical Address* is loaded on the **`%CR3`** *Register*
  - The Processor calculates the Addresses of the inner *Page Tables* and *Pages* using this
  - Remember: Each *Process* has its own *Page Directory*, therefore while *Task Context Switching* the **`%CR3`** is updated with the *Physical Address* of the *Page Directory* of the next *Process* in the list
    - i.e. the inner *(Secondary) Page Tables* are not even part of the *Process*' PCB data structure, they are setup and maintained by the OS

Efficient Translations

Our original *Page Table* approach already carries 2x the cost of *Memory* access

- One indirection into the *Page Table*, another indirection to the actual *Physical Address*

Now *Two-Level Page Tables* 3x the cost!

- 2 indirections into the *Page Directory* & the *(Secondary) Page Table,* 1 more to the actual *Physical Address*
- Worse, 64-bit Architectures (e.g. x86-64) support *4-Level Page Tables*
- And this assumes that the target *Page Table* is already in *Memory* (i.e. not previously "*Paged-Out*")

How can we use *Paging* but also reduce Lookup cost?

- Cache **Translations** in Hardware

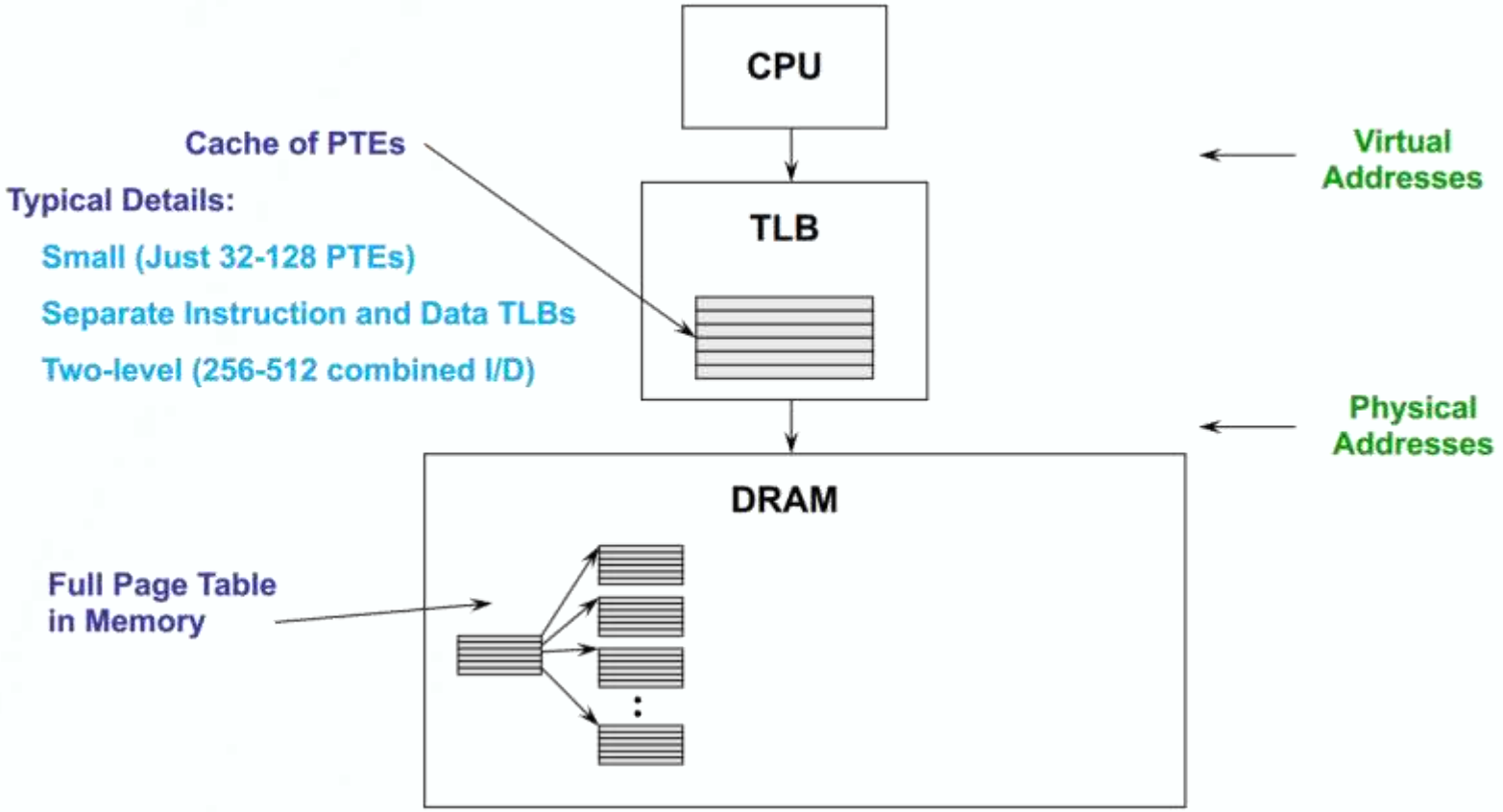
- The *Translation Lookaside Buffer* (TLB)
  - TLB managed by *Memory Management Unit* (MMU)

Virtual Memory

Translation Lookaside Buffer

- Translates a *Virtual Page Number* into a *Page Table Entry* (*Remember*: PTE is PPN + *Protection* bits)
  - Not to *Physical Addresses*, this is done by the MMU
  - Translation can be done in a single machine cycle
- TLBs implemented in Hardware
  - Typically 4-Way to Fully-Associative Memory (i.e. all entries Looked-up **in parallel**)
    - Non-sequential search, Extremely fast, but Expensive
  - Cache Keys/Tags are *Virtual Page Numbers* (VPNs)
  - Cache Values are *Page Table Entries* (PTEs)
  - With PTE + *Offset*, the MMU can directly calculate *Physical Address*
- Intuition: TLBs exploit *Locality*
  - *Processes* only use a handful of *Pages* at a time
  - 32 - 128 entries out of all of the *Process' Pages* (128 KiB – 512 KiB)
  - Only need those *Pages* to be present and "mapped" in the TLB Cache
  - Hit rates are very important

Translation Lookaside Buffer



TLB Management

- Address Translations for most *Instructions* are handled using the TLB
  - >99% of Translations, but misses do happen ("TLB *Miss*")

Who loads Translations into the TLB?

- Hardware-managed (*Memory Management Unit*) [x86]
  - Knows where *Page Tables* are in *Main Memory*
  - OS responsible for maintaining *Page Tables*, and Hardware accesses them directly
  - *Page Tables* have to be in Hardware-defined format
    - Inflexible
- Software-managed TLB (OS) [MIPS, Alpha, Sparc, PowerPC]
  - TLB *Faults* to the OS, OS finds appropriate *Page Table Entry*, loads it into the TLB
  - Must be fast (but still 20-200 cycles)
  - CPU *Instruction Set Architecture* (*ISA*) offers *Instructions* for manipulating TLB
  - *Page Tables* can be in format convenient for OS
    - Flexible

TLB Management

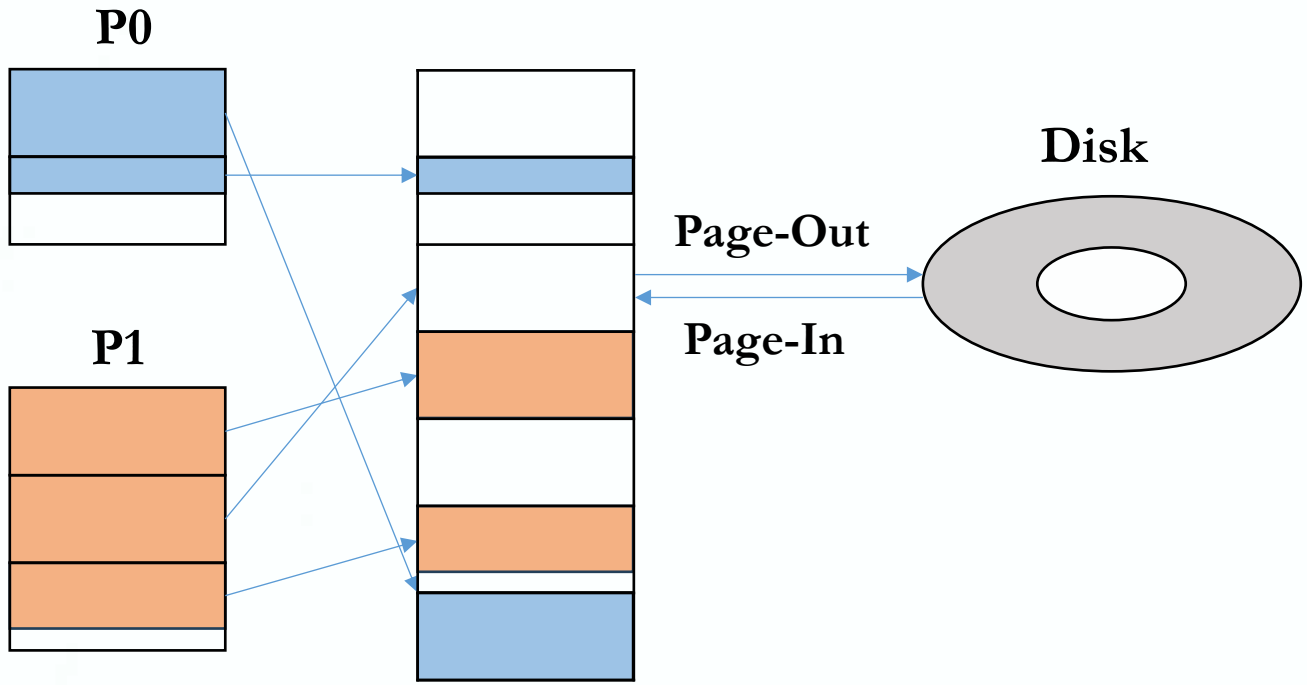
- OS responsible to ensure that TLB and *Page Tables* are **consistent**
  - When it changes the *Protection* bits of a PTE, it needs to *Invalidate* the PTE if it is in the TLB
- On *Process Context Switch* the TLB needs to be *Reloaded*
  - *Invalidate* (almost) all entries - Otherwise CPU would continue using the old Translations
    - e.g. **`invlpg`** (CPU determines *Page* that contains Address and flushes all TLB entries for that *Page* including the Global TLB mappings –and optionally a specific *Process Context IDentifier (PCID)*)
  - But Kernel-mapped *Pages* are Global, i.e. do not need to be *Invalidated*
    - e.g. **`invpcid`** (Control through *INVPCID Type* over invalidating: 0 – individual Address & specific PCID & not Global, 1 – All Addresses & specific PCIDs & not Global, 2 - All & Global, 3 – All & not Global)

When the TLB "*Misses*" and a new *Page Table Entry* has to be loaded:

- A Cached *Page Table Entry* must be "evicted"
- Choosing which *Page Table Entry* to "evict" is called the "*TLB Replacement Policy*"
- Implemented in Hardware, often simple algorithmically
  - e.g. *Least-Recently-Used* (more later on *Eviction* Policies)

Paged Virtual Memory

- *Pages* can be moved between *Memory* and Disk
  - Use Disk to simulate larger *Virtual Memory* than physically available
  - This *Process* is called "*Paging-In*" / "*Paging-Out*"



Virtual Memory

Paged Virtual Memory

- *Pages* can be moved between *Memory* and Disk
  - Use Disk to simulate larger *Virtual Memory* than physically available
  - This *Process* is called "*Paging-In*" / "*Paging-Out*"
- *Paging* a *Process* over time
  - Initially, *Pages* are allocated from *Memory*
  - When *Memory* fills up, allocating a *Page* requires some other *Page* to be evicted
  - Evicted *Pages* go to Disk – "*Swap*" file / backing storage
  - Done by the OS, and transparent to the Application
- Extreme design: *Demand Paging*
  - "*Paging-In*" a *Page* from Disk into *Memory* only if an attempt is made to access it
  - *Main Memory* acts as a "Cache" for the Disk

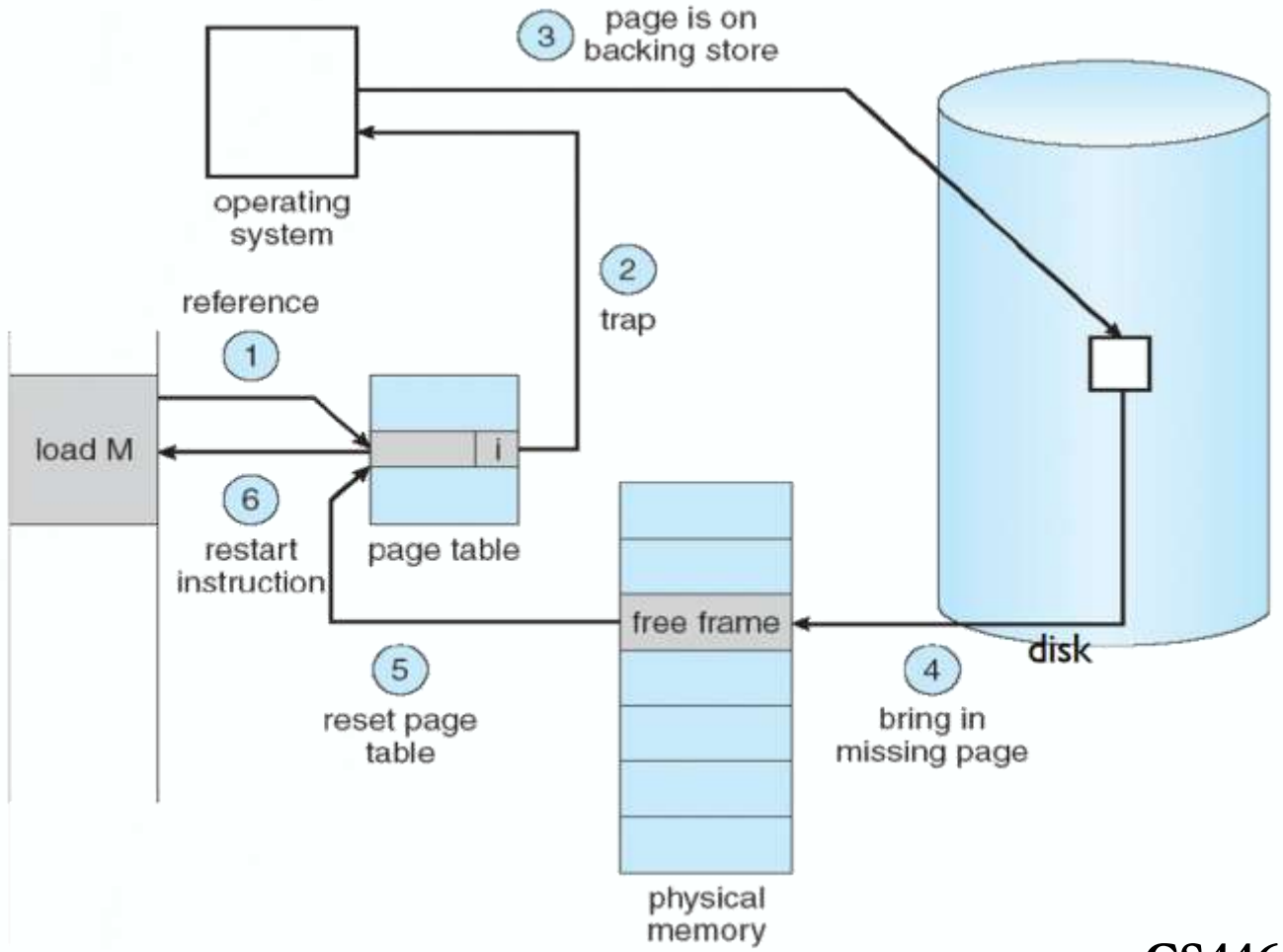
Page Faults

What happens when a *Process* accesses a *Page* that has been evicted?

- 1. When the OS evicts a *Page*, it sets the *Page Table Entry* as *Invalid* (modifies *Valid* bit) and modifies the location of the *Page* to point to the *Swap* file (location where it was "*Paged-Out*")
- 2. When a *Process* accesses that *Page*, the *Invalid Page Table Entry* access causes a *Trap*
  - *"Page Fault"*
- 3. The *Trap* is handled by the OS *Page Fault Handler*
- 4. Handler uses the *Invalid Page Table Entry* to locate *Page* in *Swap* file
- 5. Reads *Page* into a *Physical Memory* frame, updates *Page Table Entry* to point to that
- 6. Restart *Process' Instruction* that caused the *Fault*

- But where does it put it? Have to evict something else…
  - OS usually keeps a *Pool* of *Free Pages* around so that allocations do not immediately cause evictions

Page Faults


page is on backing store
3
operating system
2
trap
reference
1
load M
i
6
restart instruction
page table
free frame
5
reset page table
4
bring in missing page
disk
physical memory

All together – the Most Common case

A simple situation: *Process* is executing on the CPU, and it issues a read to an Address

The read goes to the TLB in the MMU

- 1. TLB performs a Lookup using the *Virtual Page Number* of the Address
- 2. Common case is that the *Virtual Page Number* finds a TLB match, and returns a *Page Table Entry* for the mapping for this Address
- 3. TLB validates that *Protection* bits of *Page Table Entry* allow reads (for this particular example)
  - If it would fail, *"Protection Fault"*
- 4. *Page Table Entry* specifies which *Physical Frame* holds the *Page*
- 5. MMU combines the *Physical Frame* + *Offset* into a *Physical Address*
- 6. MMU then reads from that *Physical Address*, returns value to CPU

Note: This is all done by the Hardware

TLB *Misses* and *Protection Faults*

Two other things can also happen:

- *TLB Miss* :
  - TLB does not contain a Cached *Page Table Entry* mapping this *Virtual Address*
- *Protection Fault* :
  - PTE in TLB, but *Memory* access violates PTE *Protection* bits

TLB *Misses* and *Protection Faults*

Reloading the TLB

If the TLB does not contain the Mapping, two possibilities:

- 1. MMU Hardware loads *Page Table Entry* from *Page Table* in *Memory*
  - Hardware-managed TLB, OS not involved in this step
  - OS has already set up the *Page Tables*, so that the Hardware can access it directly
- 2. Trap to the OS – Software-managed TLB
  - OS intervenes & performs Lookup in *Page Table*, loads the *Page Table Entry* into TLB
    - *Remember: Page Directory* first, then *Page Table*, but *Page Table* might have been "*Paged-Out*", so it needs to be brought back in (yet another *Trap*); this process can indeed become quite complicated…
  - OS returns from Fault, TLB continues
- An Architecture will only support one method or the other

TLB *Misses* and *Protection Faults*

Reloading the TLB

Page Table Lookup (by HW or OS) can cause recursive *Fault* if *Page Table* has been *"Paged-Out"*

- Assuming *Page Tables* themselves are in OS part that is Mapped to *Virtual Address Space*
- Not a problem if all *Page Tables* were *"Wired"* to *Physical Memory*
  - e.g. like *Page Directories* are

When TLB has *Page Table Entry* it restarts Translation

- Common case: *Page Table Entry* refers to a *Valid (/Present) Page* in *Memory*
  - These Faults are handled quickly, just read *Page Table Entry* (from the *Page Table* which is in *Memory*) and load it into TLB
- Uncommon case: TLB *Faults* again on *Page Table Entry* because of its *Protection* bits
  - e.g. *Page* is *Invalid* (/*Not Present* in *Memory*)
    - becomes a *Page Fault*

TLB *Misses* and *Protection Faults*

Protection Faults

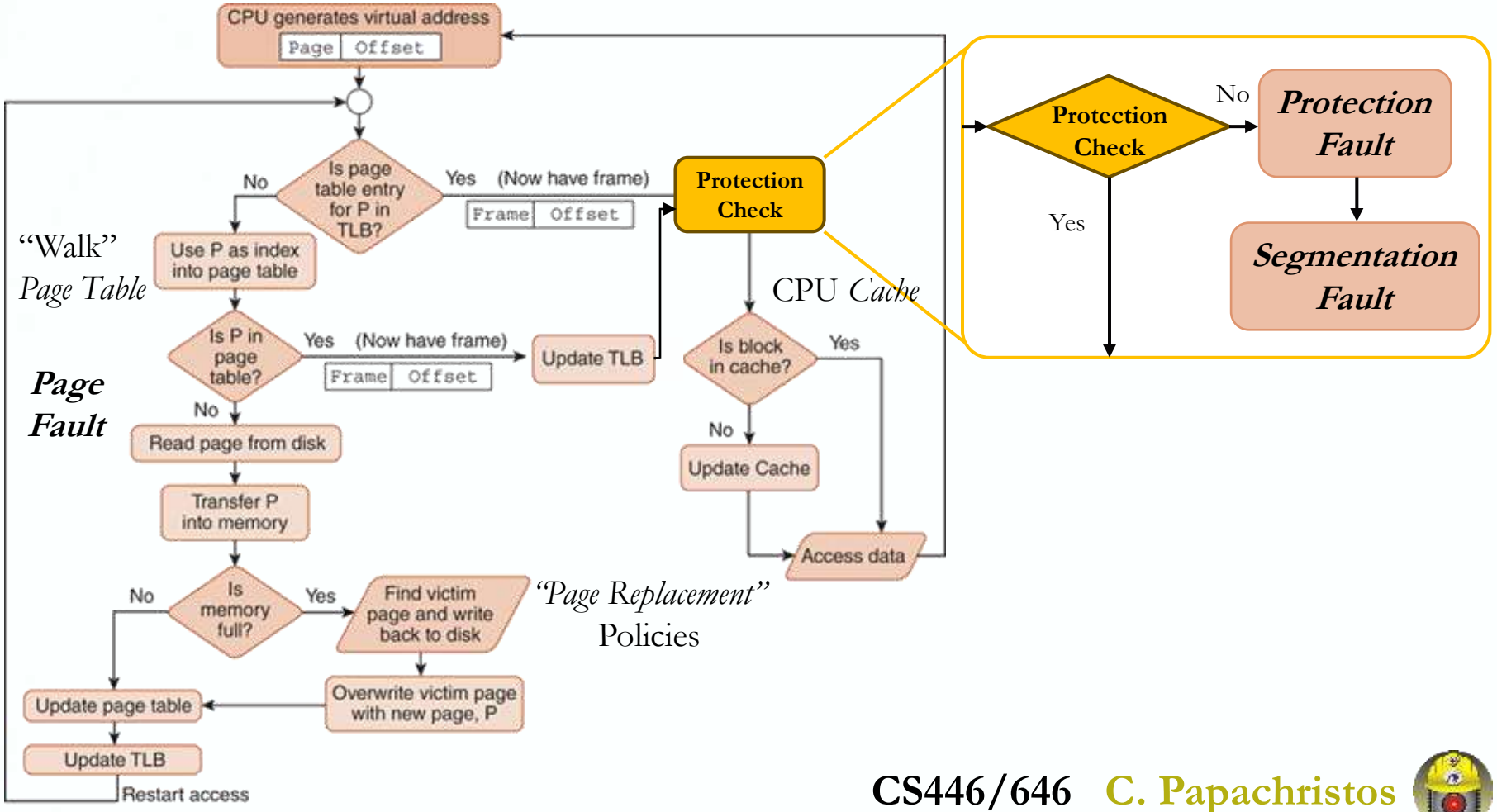
Page Table Entry can indicate a *Protection Fault*

- *Read/Write/Execute* bits – Operation not permitted on *Page*
- *Invalid* bit – *Page* not allocated (for *Process*), or *Page* not present in *Physical Memory* ("*Paged-Out*")

TLB *Traps* to the OS (Software takes over)

- R/*W*/*E* – *"Protection Fault"* : OS usually will send *Fault* back up to *Process*
  - or might be part of "trick" implementations (e.g. *Copy-on-Write*, *Mapped Files*, more later)
- *Invalid* – *"Page Fault"*:
  - *Virtual Page* not allocated in *Address Space* of *Process*
    - OS sends fault to *Process* (e.g. *Segmentation Fault*)
  - *Page* not present in *Physical Memory*
    - OS allocates *Physical Frame*, reads from Disk, maps *Page Table Entry* to *Physical Frame*

TLB *Misses* and *Protection Faults*



Virtual Memory

Advanced Functionalities through *Virtual Memory* "Tricks"

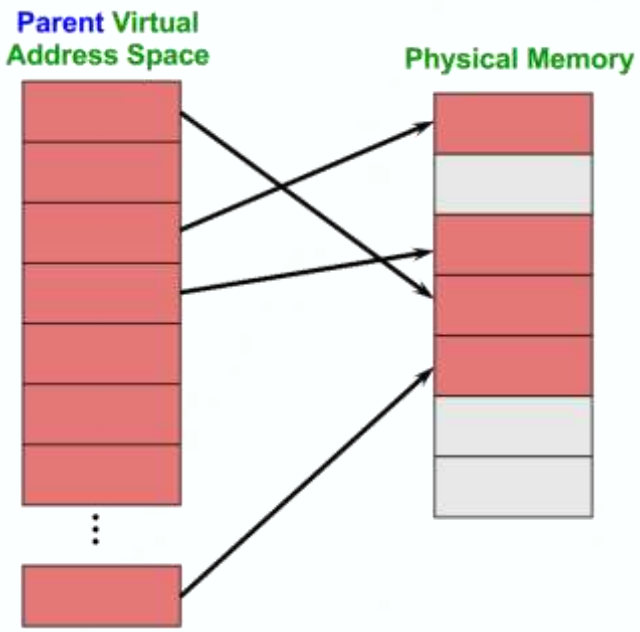
- *Copy-on-Write*
- *Memory-Mapped Files*
- *Shared Memory*

Copy-on-Write

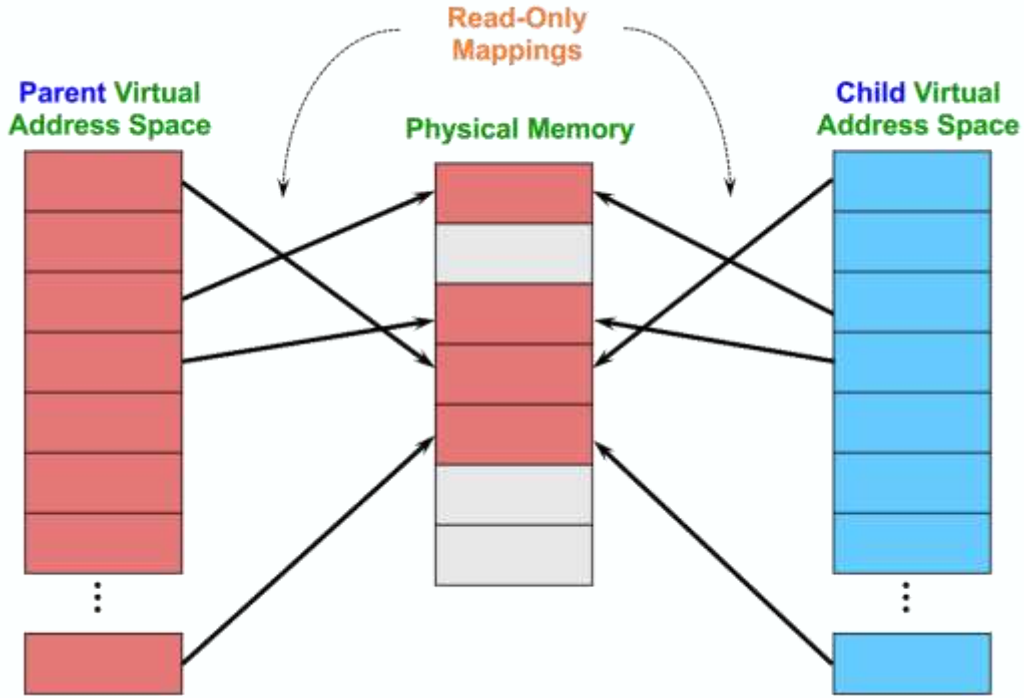
- OSs spend a lot of time copying data
  - *System Call* arguments between *User* and *Kernel Space* (for ***Security*** & ***Reliability***)
  - Entire *Address Spaces* to implement `fork()` – (in theory, let's see why not…)
- Use *Copy-on-Write* (*CoW*) to defer large copies as long as possible, hoping to avoid them altogether
  - Instead of copying *Pages*, create *Shared* mappings of *Parent Pages* in *Child*'s *Virtual Address Space*
  - *Shared Pages* are *Protected* (*Page Table Entry* bit)as **read-only** in *Parent* **and** *Child*
  - Reads happen as usual, but a write generates a *Protection Fault* → *Trap* to OS → Copy *Page* → Change *Page* Mapping in *Child*'s *Page Table* → Restart write *Instruction*
- What `fork()` actually does…

Copy-on-Write

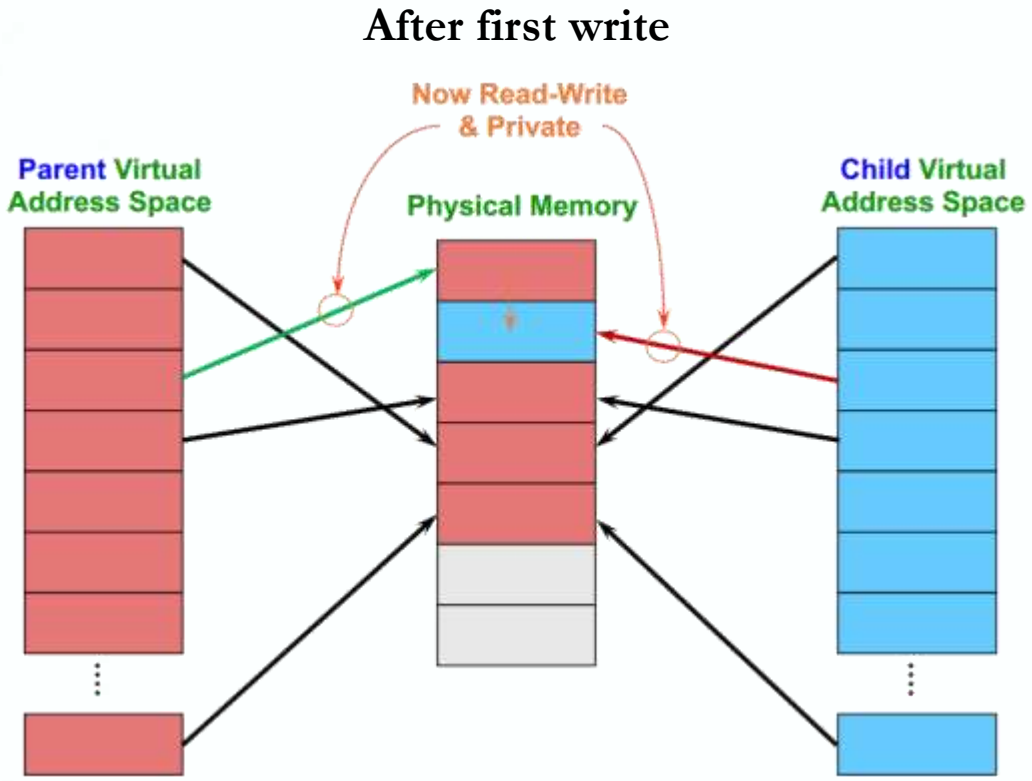
Before `fork()`



After `fork()`



Copy-on-Write



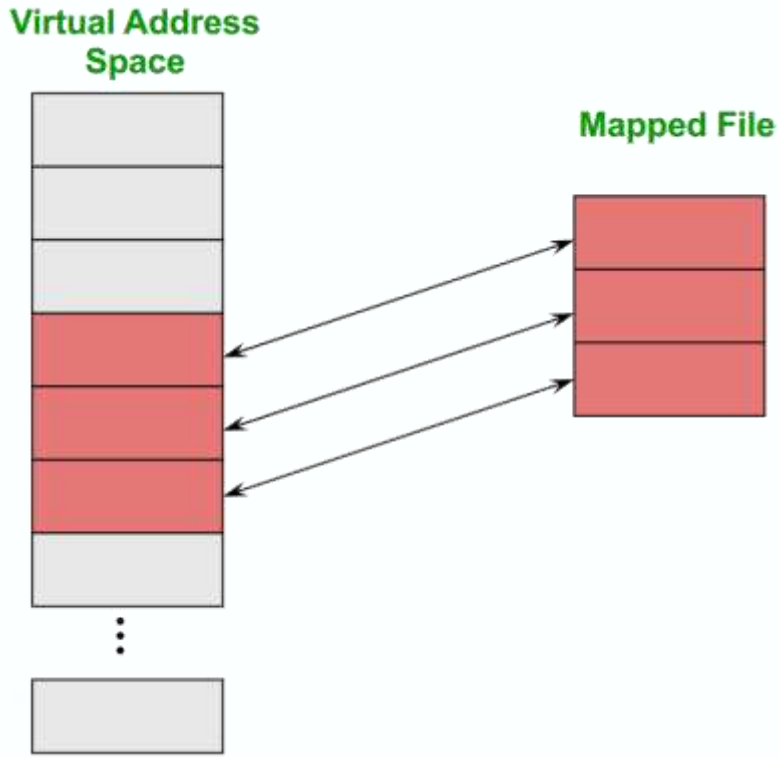
Virtual Memory

Memory-Mapped Files

- *Memory-Mapped* Files enable *Processes* to do *File* I/O using **`load`**s and **`store`**s
  - Instead of *System Calls*, e.g. **`open()`**, **`read()`**, **`write()`**, **`fseek()`**
- Bind a *File* to a *Virtual Memory Region*
  - **`mmap()`** in Unix (https://linux.die.net/man/2/mmap)
  - *Page Table Entries* Map *Virtual Addresses* to *Physical Frames* holding *File*'s data
  - *Virtual Address* : *Base* + *Offset* refers to *Offset* in *File*
- Initially, all *Pages* Mapped to a File are *Invalid*
  - OS reads a *Page* from *File* when *Invalid Page* is accessed (*Page Fault Handling*)
  - OS will only write a *Page* to *File* when it is *Evicted*, or region *Unmapped* (**`munmap()`**)
  - If *Page* is not *Modified*/*Dirty* (not written-to), no write to *File* is needed at all
    - This is another use of the *Dirty* bit in *Page Table Entries*

Memory-Mapped Files

File-backed Virtual Memory



Virtual Memory

Memory-Mapped Files

- *File* essentially acts as backing store for that region of the *Virtual Address Space* (similar concepts to using the *Swap* File)
  - *Virtual Address Space* not backed by "real" *Files* also called *Anonymous Virt Memory*

Advantages

- Uniform access for *Files* and *Memory* (can just use Pointers as the interface)
- Less copying (write-to-*File* only on *Page Eviction*, *Unmapping*)
  - Similar motivation as with *Copy-on-Write*

Drawbacks

- *Process* has less control over data movement (what would happen in a system "crash"?)
  - OS handles *Faults* transparently
- Does not generalize to streamed I/O (*Pipes*, *Sockets*, etc.)

Virtual Memory

Memory-Mapped Files

- *File* essentially acts as backing store for that region of the *Virtual Address Space* (similar concepts to using the *Swap* File)
  - *Virtual Address Space* not backed by "real" *Files* also called *Anonymous Virt Memory*

Also very importantly:

- Facilitate *Shared Memory* & *Inter-Process Communication (IPC)*
  - Several *Processes* can *Memory-Map* the same *File*
    - Allows *Pages* in *Memory* to be *Shared*
  - *Memory Persistence*
    - Enables *Processes* to *Share Memory* sections that **persist** independently of the lifetime of a certain *Process*

Virtual Memory

Shared Memory

- Private *Virtual Address Spaces* protect Applications from each other
  - Usually exactly what we want to accomplish
- But data sharing might also be desired
  - *Parents* and *Children Processes* in a `fork()`ing Web server or proxy will want to *Share* the same in-*Memory* cache
- We can use *Shared Memory* to allow *Processes* to share data using direct *Memory* references
  - Both *Processes* will be able to see updates to the *Shared Memory Segment*
  - *Process* B can immediately read an update by *Process* A
  - *Note:* Requires *Synchronization* through *Shared* objects performing some *Synchronization Mechanisms*

Shared Memory

Implement *Sharing* using *Page Tables* :

- Have *Page Table Entries* in both *Page Tables* map to the same *Physical Frame*
- Each *Page Table Entry* can have different values for its *Protection* bits
- Must update both *Page Table Entries* when *Page* becomes *Invalid*

Can map *Shared Memory* at same or different *Virtual Addresses* in each *Process' Address Space*

- Different: Flexible (no *Address Space* conflicts), but the **values** of Pointers inside the *Shared Memory Segment* are (most probably) invalid (Why?)
- Same: Non-portable, but **values** of Pointers inside *Shared Memory* are valid

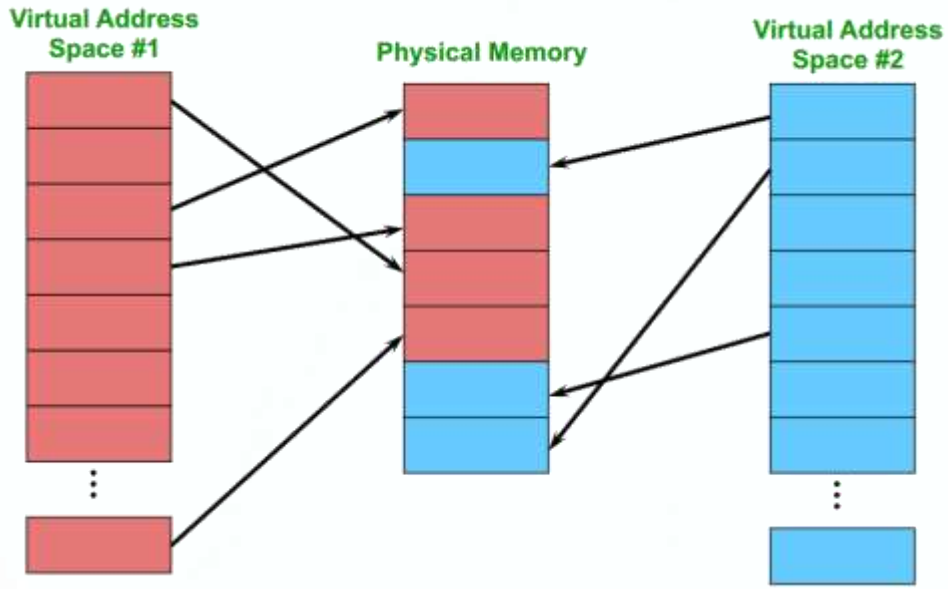
```
typedef struct {
   int * arr_p;
   int arr[256];
} data_t;
```

```
inst shmid = shmget(KEY, sizeof(data_t),
                          IPC_CREAT | 0666);
data_p = (data_t *)shmat(shmid);
data_p->arr_p = data_p->arr;  //Problem !
```

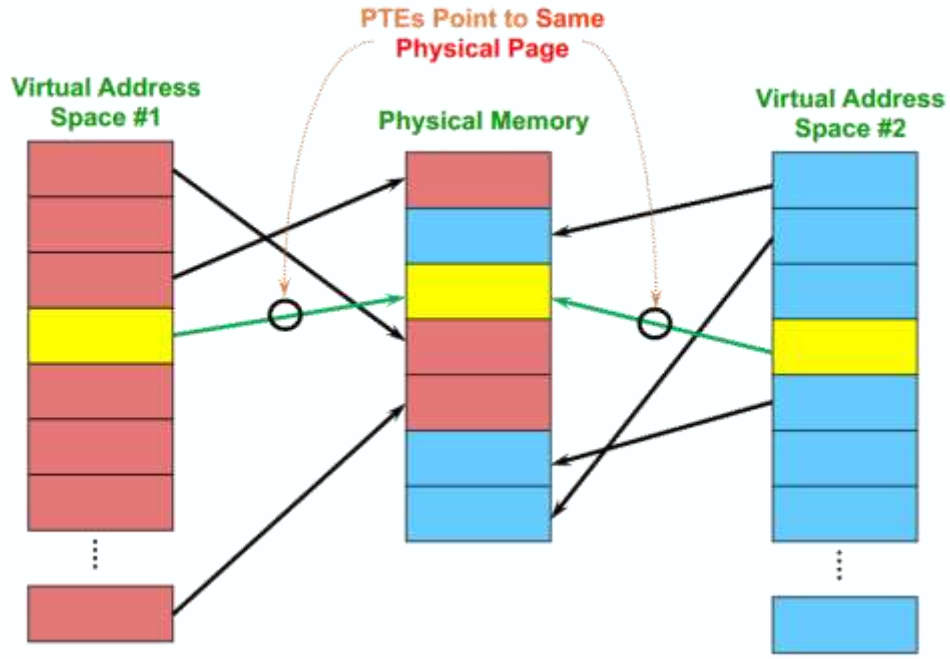
- *Also Caution:* Pointer in *Shared Memory Segment* referencing Address outside the *Shared Memory Segment*

Shared Memory

Isolation: No Sharing



Sharing Pages



CS-446/646

Time for Questions !