

# Virtual Memory

## ¿Que problemas llevaron a esta ?

- No suficiente memoria (RAM)
- Huecos en el address space (Solo se podían colocar los procesos de manera completa en el address space)
- Override de un proceso encima de otro si necesitaban el mismo address space

## ¿Que es Virtual memory?

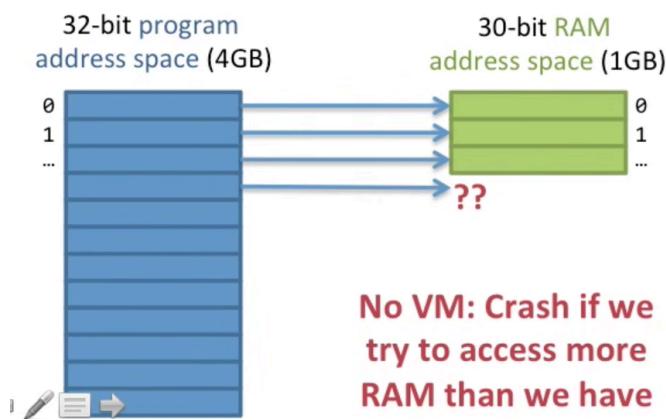
- Es una manera de redireccionar hacia el address space.  
- La palabra clave del problema es que los procesos comparten **el mismo espacio de memoria**

### • Solución

- Cada proceso va a tener su propia memoria virtual (Aka: Mapping) significa que cada proceso va a tener 3 cosas
  1. Una memoria virtual por programa
  2. Un map que apunta la dirección virtual al verdadero address space
  3. El memory address space (RAM), que también puede llegar a ser el disco

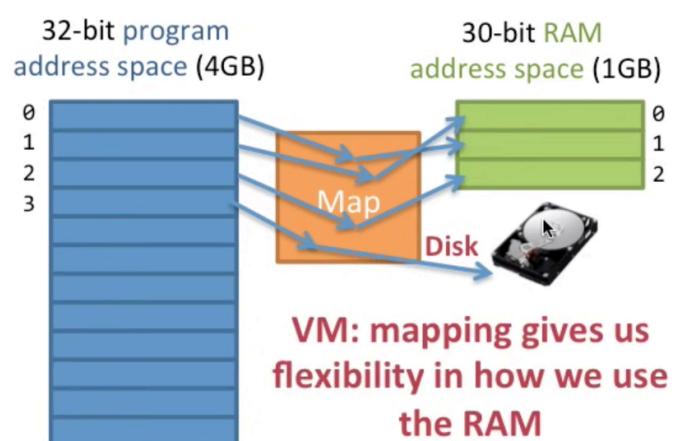
## Without Virtual Memory

Program Address = RAM Address



## With Virtual Memory

Program Address Maps to RAM Address

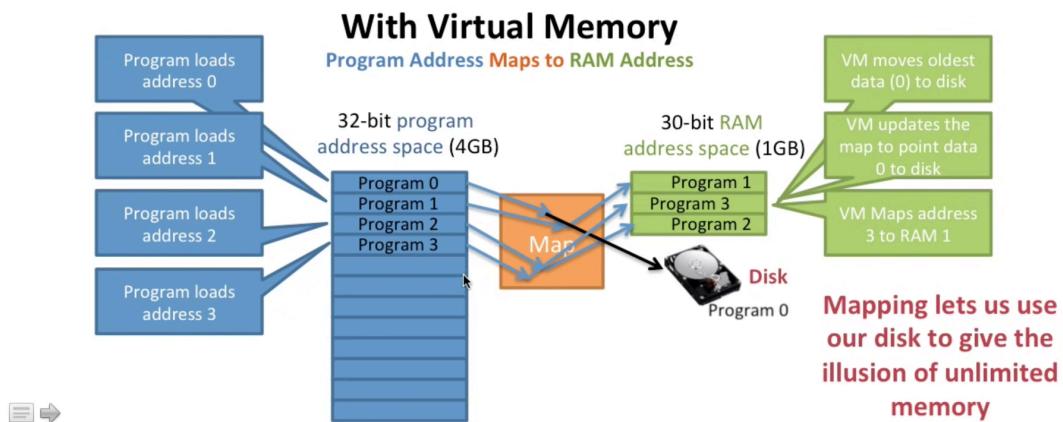


- **Mapping**

Esta es la habilidad que cuando la RAM esté full se pueda pasar de la RAM al disco duro haciendo un swap

**Map** some of the program's address space to the **disk**

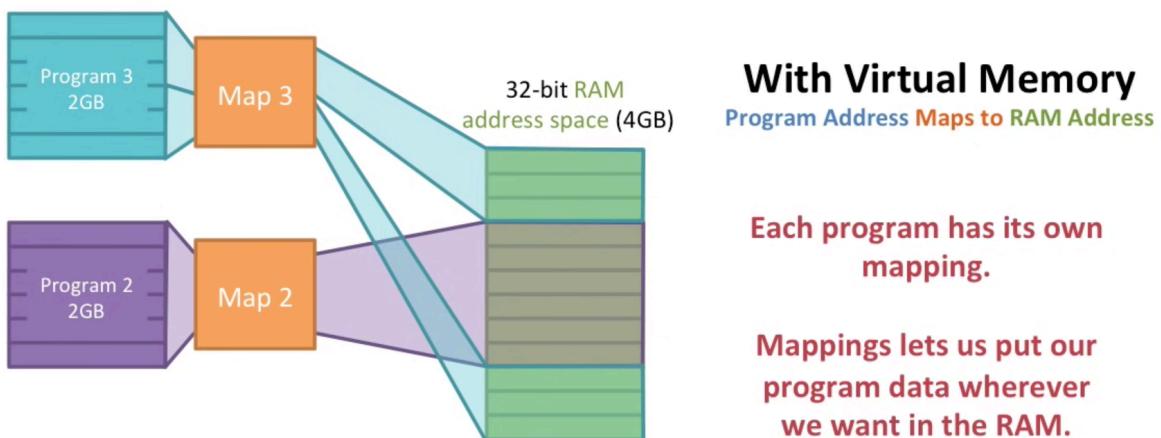
When we need it, we bring it into memory



También con el mapping se puede tener múltiples procesos corriendo y evitar los huecos que se generaban así se usa toda la RAM de manera efectiva.

Además elimina el error de que un programa borre o sobre escriba un valor en el mismo espacio de memoria. Y para compartir memoria cada map si tendría que compartir el mismo address space (No necesario para nachos).

- How do we use the holes left when programs quit?
- We can **map** a program's addresses to **RAM addresses** however we like



## • ¿Cómo funciona?

Básicamente se tiene dos conceptos

1. **Virtual memory:** Es lo que el proceso ve
2. **Physical memory:** El espacio físico de la computadora (**RAM**)

Entonces se tienen estos dos ideas

### 1. Virtual Addresses (VA)

- Lo que el programa usa
- En mips es un total de 32-bit address space. (Cree tener disponible la ram, no se cómo funciona en Nach Os)

### 2. Physical Addresses (PA)

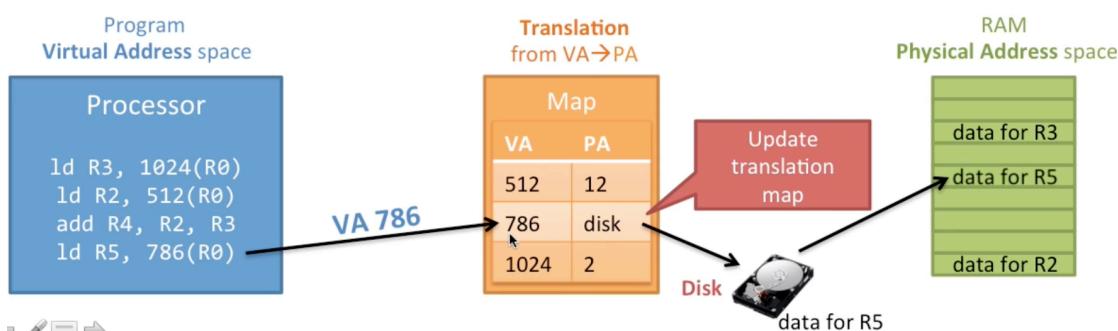
- Lo que usa el hardware para hablar con la RAM
- El address space es determinado por la cantidad de RAM que hay instalada

## • ¿Cómo un proceso accede a la memoria?

1. El proceso carga usando el VA
2. El OS traduce el address (VA) al physical address (PA) en la memoria principal
3. Si el proceso no está en la memoria principal entonces el OS lo carga del disco
4. El OS lee la RAM usando la PA y retorna la data

### How does a program access memory?

1. Program executes a load with a **virtual address (VA)**
2. Computer **translates** the address to the **physical address (PA)** in memory
3. (If the **physical address (PA)** is not in memory, the operating system **loads it in from disk**)
4. The computer then **reads the RAM** using the **physical address (PA)** and returns the data to the program

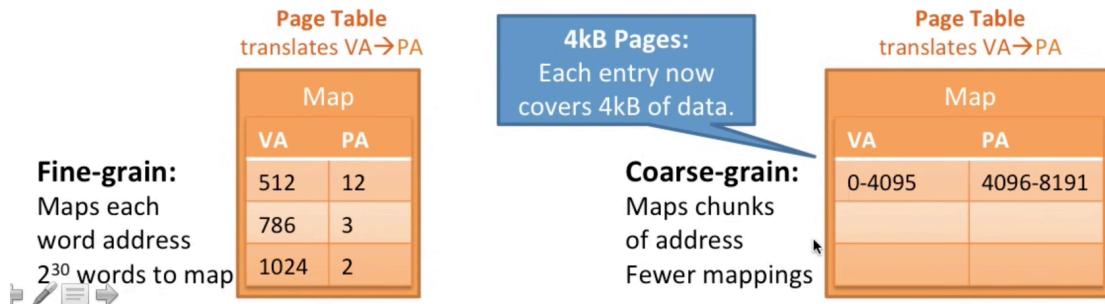


## • Page tables (Keeping track of VA -> PA mappings)

El map que tienen los procesos es lo que se llama el **Page table**, el punto es que por cada puntero que teníamos del VA por el PA hacia que este **Page Table** fuera muy problemas entonces en vez de tener un VA que apunta cada una de las distintas direcciones se maneja por **Page** esto hace que sea mucho más pequeño el **mapping**, y se maneja en chunks de la dirección 0 a la 4095, osea usando un **page table** de 4KB

## Page table size

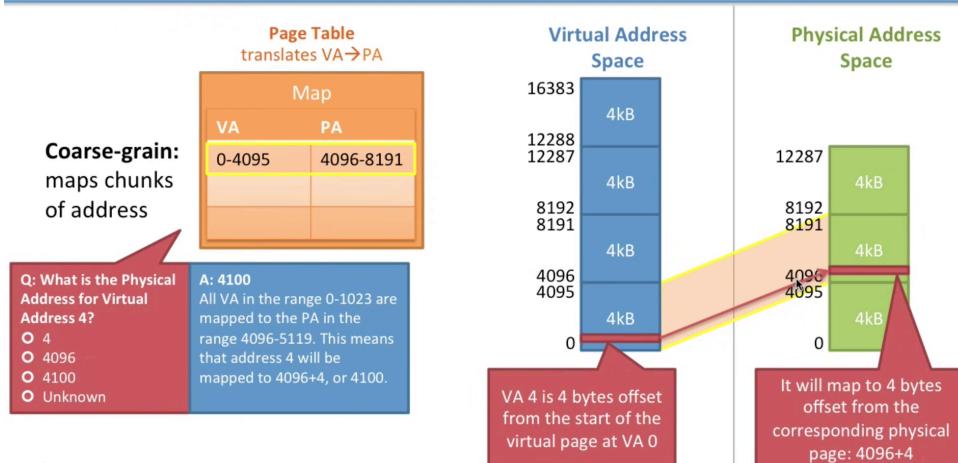
- We need to translate every possible address:
  - Our programs have 32-bit Virtual Address spaces
  - That's  $2^{30}$  words that need Page Table Entries (1 billion entries!)
  - (If they don't have a Page Table Entry then we can't access them because we can't find the physical address.)
- How can we make this more manageable?
  - What if we divided memory up into chunks (**pages**) instead of words?



Que problemas puede presentar esto, es que es menos flexible, antes se podía mover **word by word** ahora eso no es posible se tiene que mover todo el chunk de direcciones. Se pierde flexibilidad a cambio de poder almacenar memoria por esta **Page Table**.

A partir de ahora se piensa en los VA por **pages**, ya no por **words**. Igual para el PA se piensa en paginas

## How do we map addresses with pages?



## • Address Translation

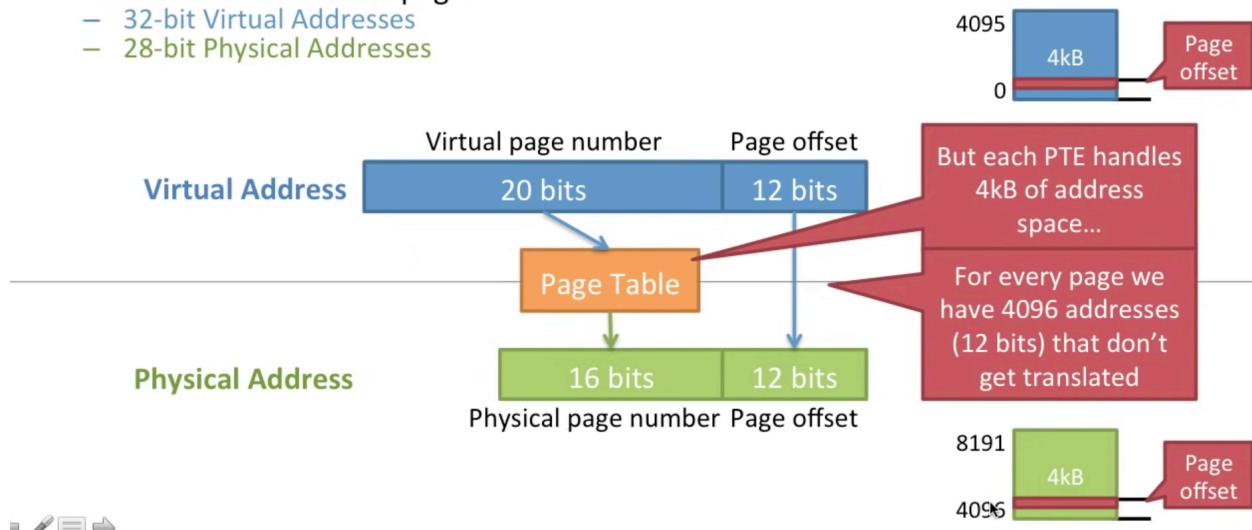
Cómo se traduce VA en PA usando page tables, a modo de resumen es que si tiene las páginas de 4kb pero se tiene un offset porque se tienen que saber dos cosas:

1. Que página (Page)
2. En qué parte de la página (Offset)

El offset se calcula según la cantidad de addresses (direcciones) que hay en una página entonces si la página es de 4KB = 4096 addresses es igual que decir 12 bits.

Este offset no se tiene que traducir ya que es el mismo por cada página dependiendo en qué parte se acceda

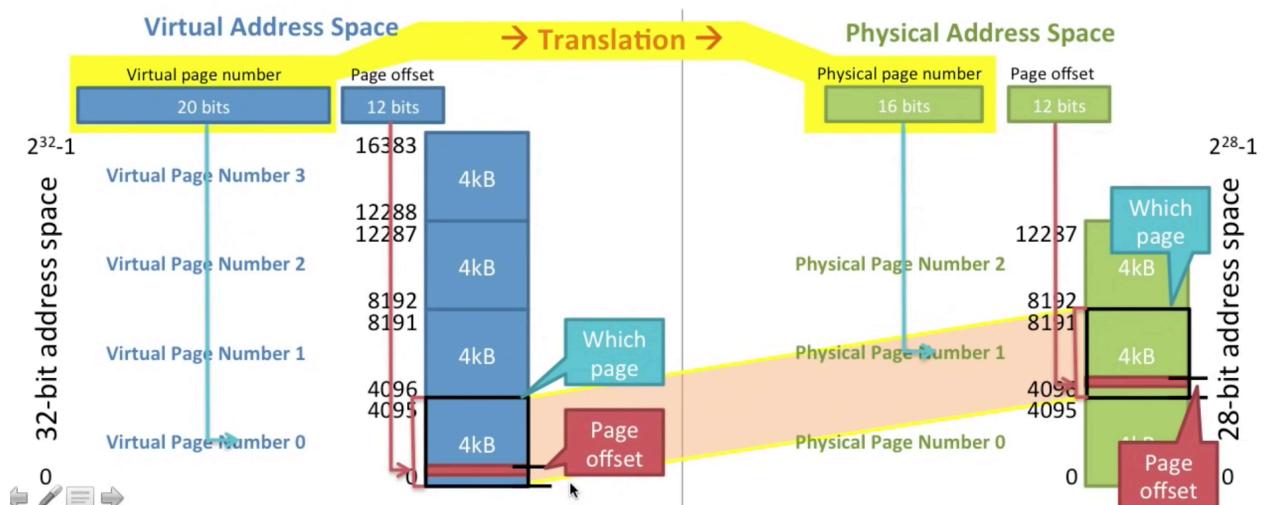
- What happens on a 32-bit machine with 256MB of RAM and 4kB pages?
  - 32-bit Virtual Addresses
  - 28-bit Physical Addresses



Un ejemplo de cómo se comporta es así

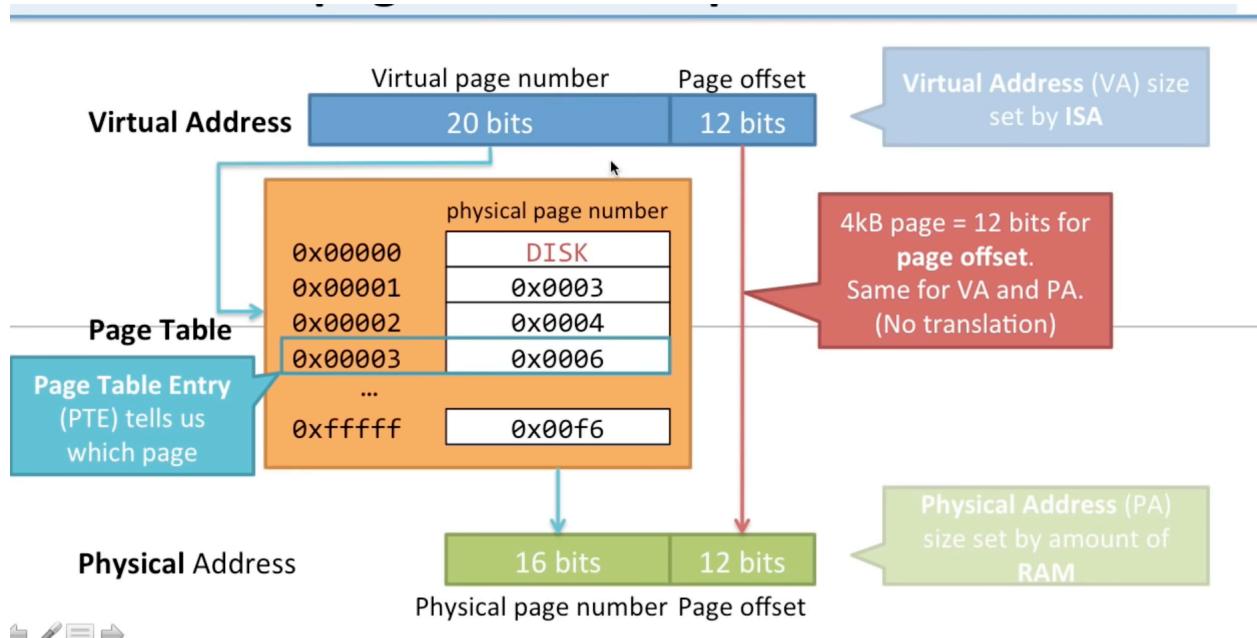
## Pages, offsets, and translation

- What happens on a 32-bit machine with 256MB of RAM and 4kB pages?

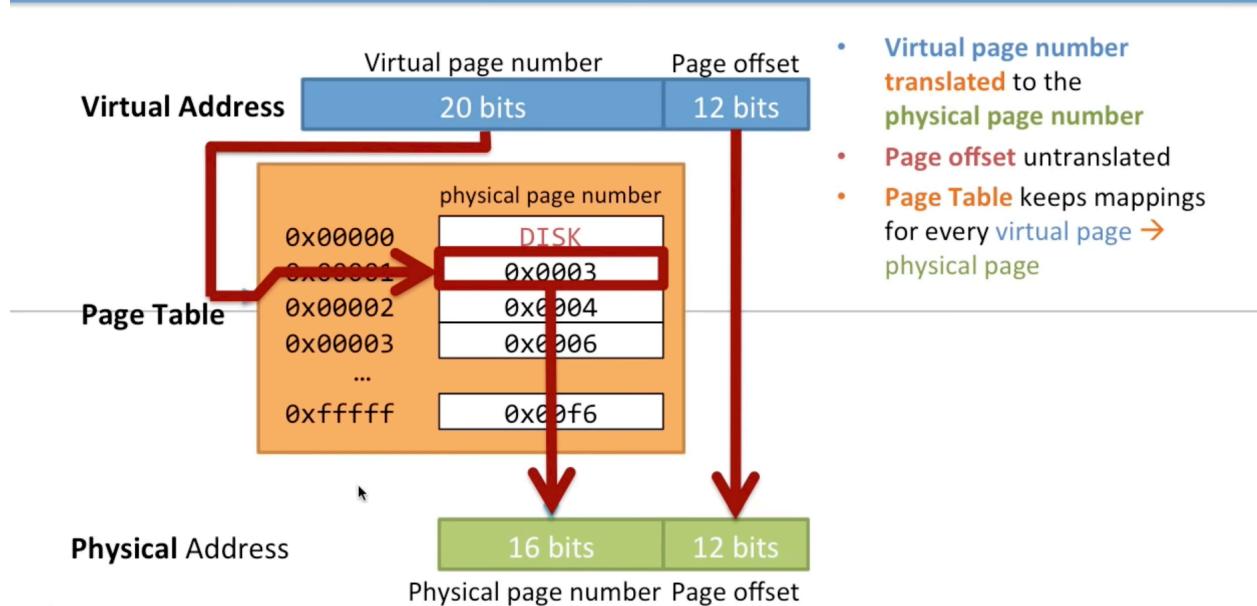


## • How works the Address Translation?

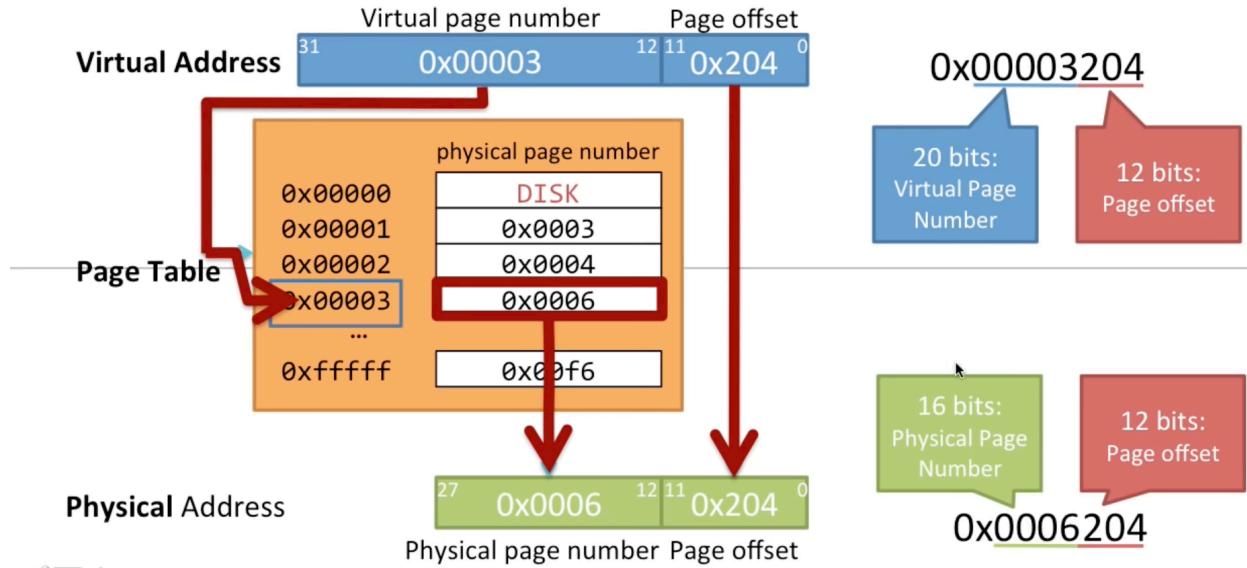
Funciona muy parecido a los words, se genera un offset dependiendo del tamaño del page table, el page table (map) tiene la dirección del VA que apunta al PA o al disco, la cantidad de bits que tenga la dirección VA depende de la cantidad bits que quedaron después del offset, igualmente sucede con los de PA



Este es un ejemplo



Junto a otro ejemplo para que se entienda mejor



- **Page Faults**

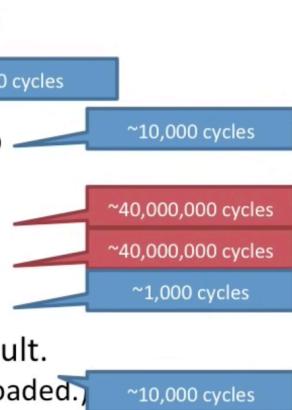
Cómo se puede saber cuando algo no esta en el PA y esta en el disco ? Esto se puede saber por la dirección de memoria, esta va a decir que esta en el disk (Revisar cómo funciona en nachos), en resumen la dirección de memoria es distinta entre la RAM y la NVRAM

- **Que sucede si la pagina no esta en RAM?**

1. La PTE (Page Table Entry) dice que la pagina esta en disco
2. El hardware (CPU) genera una excepción de tipo **page fault exception**
3. El hardware salta al OS para que maneje esta excepción
  1. OS escoge una pagina para sacarla del RAM y escribirla en el disco
  2. Si la página esta sucia necesita ser escrita en el disco primero \*\*No entiendo. Cuando una pagina es cargada a memoria RAM si esta sufre cualquier modificación (Dejo de estar igual que en su estado inicial) se marca como **Dirty**, al estar **Dirty** antes de hacer el swap se tiene que escribir en el disco para que las modificaciones no se pierdan. Si una page está **clean** entonces simplemente se puede sobreescibir por la nueva page que le va a caer encima porque si se va a recuperar solo es necesario sacar la info del proceso.
3. El OS lee la pagina y la pone en la RAM
4. El OS cambia el **Page Table** para que mapee la nueva página

#### 4. El OS salta a la instrucción que cause el page fault (Ahora ya no va a pasar porque ya está cargado en la RAM)

- Page Table Entry says the page is on disk ~1 cycles
- Hardware (CPU) generates a **page fault exception** ~100 cycles
- The hardware jumps to the OS page fault handler to clean up ~10,000 cycles
  - The OS chooses a page to evict from RAM and write to disk
  - If the page is **dirty**, it needs to be written back to disk first
  - The OS then reads the page from disk and puts it in RAM
  - The OS then changes the **Page Table** to map the new page
- The OS jumps back to the instruction that caused the page fault. ~10,000 cycles
  - (This time it won't cause a page fault since the page has been loaded.)



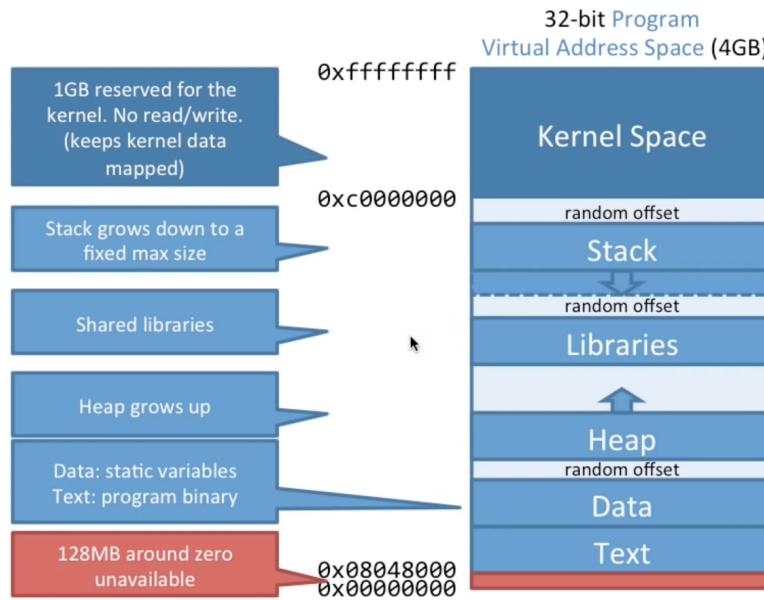
In the time it takes to do handle one page fault  
you could execute 80 million cycles on a modern CPU.

Page faults are the SLOWEST possible thing that can happen to a computer  
(except for human interaction).



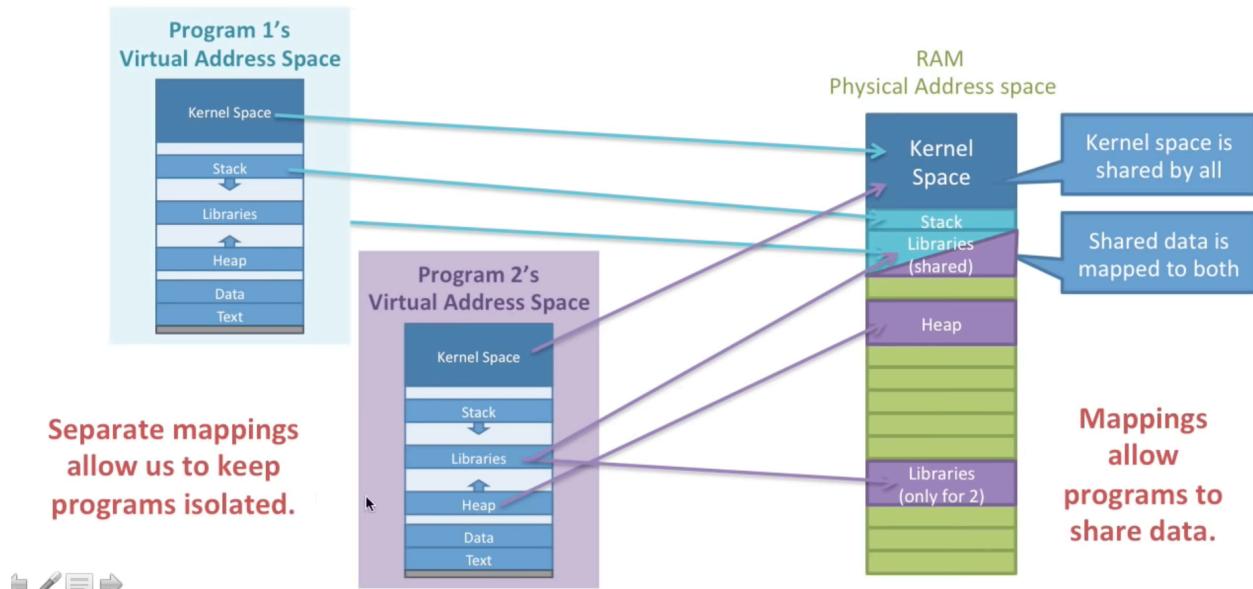
**Ejemplo:** De un proceso en linux y su división por kernel space, stack, libraries, heap, data a text. Esto es para luego presentar un ejemplo de cómo el **Virtual memory** funciona con un proceso como tal:

- Each program has its own 32-bit virtual address space
- Linux defines how that address space is used:
  - 1GB reserved for kernel
  - Program static data at the bottom
  - Heap grows up
  - Stack grows down (and has a fixed maximum size)
- Random offsets to enhance security
  - (You never know exactly where a certain piece of data/code will be.)



Hay multiples partes de un proceso que el **page table** comparten con el **PA** como el espacio del kernel y las librerías un ejemplo de cómo se ve el **PA** es así:

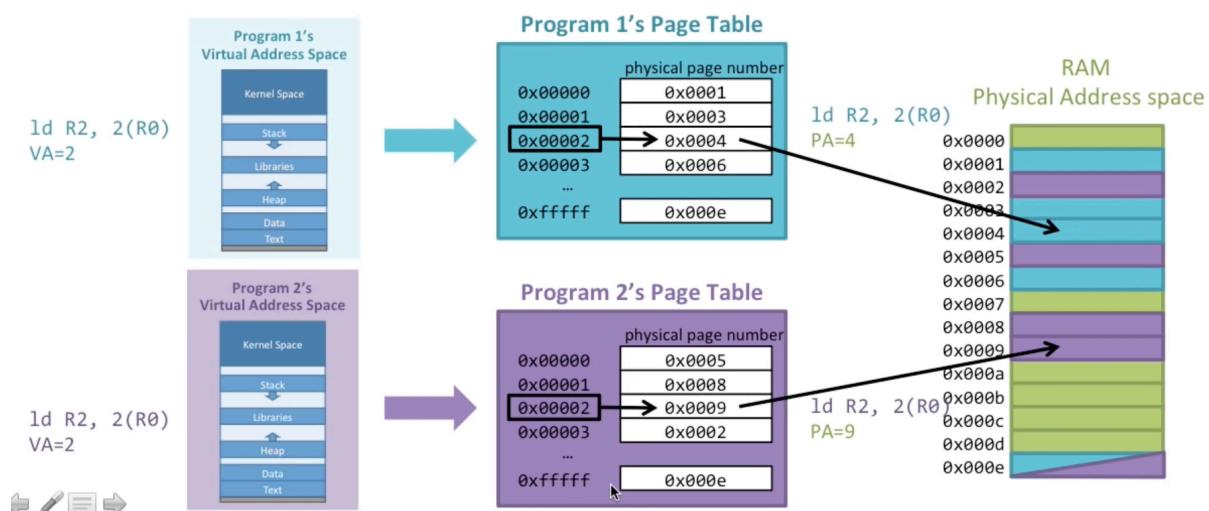
## Program address space in Linux



Otro ejemplo de la presentación es enseñando las distintas direcciones del VA

## How do we provide separate mappings?

- Each process needs its own Page Table
- OS makes sure they only map to the same physical address when we want sharing



Se ve cómo en el VA 0xffff comparten memoria

## • **TLB (Translation Lookaside Buffer)**

- Básicamente es una manera de tener cache para el VM, porque buscar en el page table puede ser muy lento de 20 a 1000 ciclos
- El TLB necesita ser pequeño y básicamente lo que hace es ser una cache, que lo que guarda son las direcciones de las últimas **Page Table** a las que se fue.
- Es un buffer para ver cual es la traducción (translation) de VA a PA

Processor Pipeline → VA → TLB → PA → Memory

- To make VM fast we add a special **Page Table cache**:

### the **Translation Lookaside Buffer (TLB)**

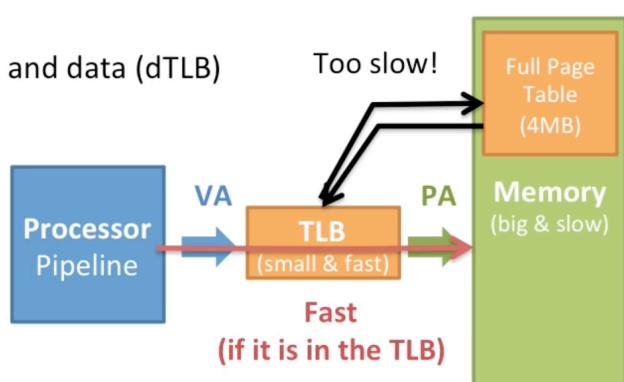
- Fast: less than 1 cycle (have to do it for every memory access)
- Very similar to a cache

- To be fast, TLBs must be small:

- Separate TLBs for instructions (iTLB) and data (dTTLB)
- 64 entries, 4-way (4kB pages)
- 32 entries, 4-way (2MB pages)
- (Page Table is 1M entries)

Lots of locality!  
Miss rates are typically  
only a few percent.

Algunas soluciones que hacen los OS modernos es hacer las paginas más grandes tener una segunda TLB más grande y tener modificaciones en le hardware, esta parte no toca en nachos ya que solo se usa una



**Q:** TLBs are small. How can we make them effectively bigger without slowing them down?

- Just make them hold more PTEs!
- Make pages larger
- Add a second TLB that is larger, but a bit slower
- Have hardware to fill the TLB automatically if there is a miss.  
(E.g., instead of having the OS do it in software.)

**A:**

### 1. Make pages larger

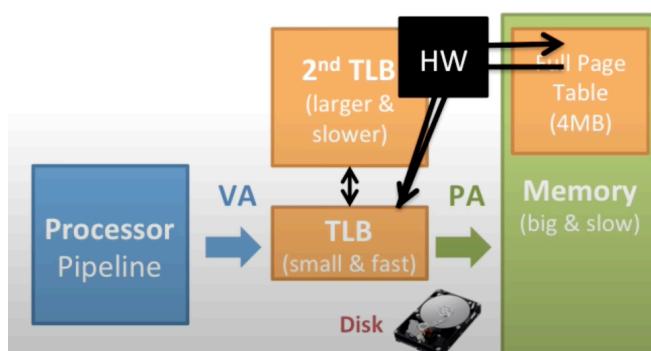
This increases the reach of the TLB because you need fewer pages to cover more data.

### 2. Add a second TLB that is larger, but a bit slower

Sure. Most processors have a level 2 TLB that is about 8x larger than the level 1 TLB, but also about twice as slow.

### 3. Have hardware to fill the TLB automatically

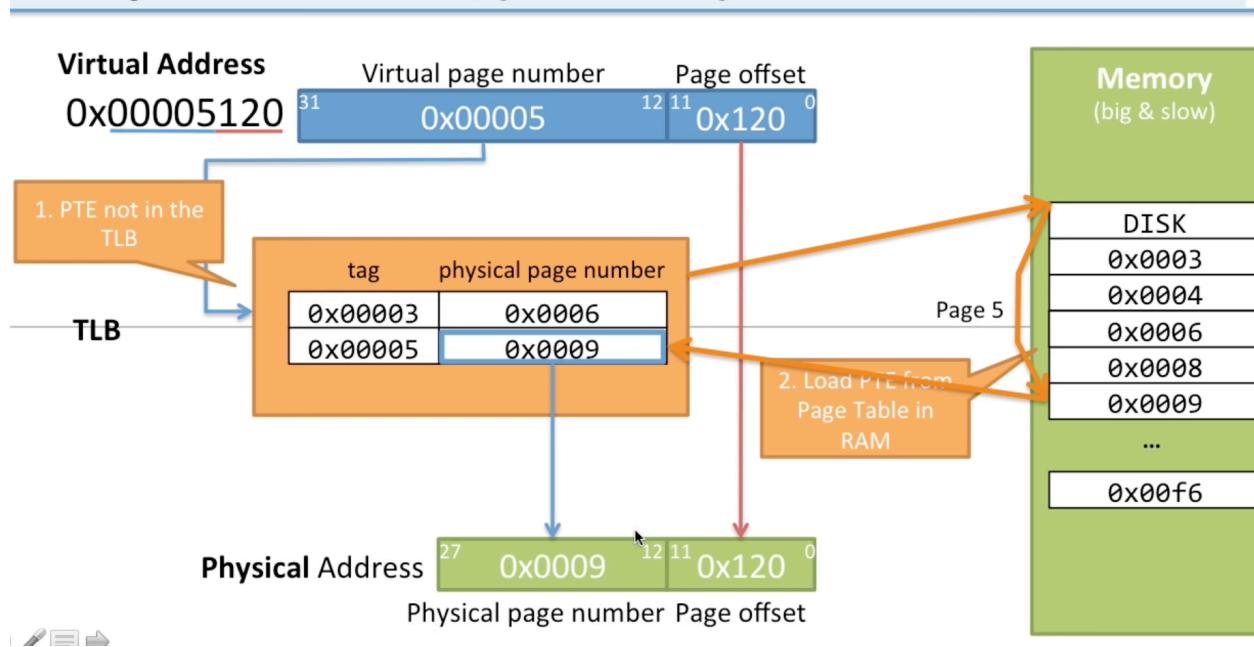
This is called a “hardware page table walk”. Basically the hardware assumes the page table is in a special form in memory, and it can go get data from it on a TLB miss without having to go to the OS.



64 4kB pages = 256kB of data  
32 2MB pages = 64MB of data

- **TLB how it works?**

- Es un cache, el punto es que ir al PTE (Page table entry) es ir hasta la ram porque siempre esta almacenado en nuestra memoria principal, ver el VA a que es igual al PA y luego cargar la verdadera dirección lo que nos permite el TLB es tener un map de estos translates que se tienen que hacer. (No entiendo exactamente donde se almacena)



- **Multi-level page tables**

- Acá esta el video no se si nos entra entonces lo deje morir

<https://www.youtube.com/watch?v=Z4kSOv49GNc&list=PLiwt1iVUi9s2Uo5BeYmwkDFUh70fJPxX&index=12>

- **Virtual cache**

- Acá esta el video no se si nos entra entonces lo deje morir

<https://www.youtube.com/watch?v=3sX5obQCHNA&list=PLiwt1iVUi9s2Uo5BeYmwkDFUh70fJPxX&index=14>