Paging

CS 241

Page Tables So Far

- **Virtual Addresses** are made up of two identifiable parts:
  - **Page Number**
  - **Page Offset**

- **Page Tables** provide translation from a **Virtual Address** to a **Physical Address**.
  - Made up of a table of **Page Table Entries** (PTEs).

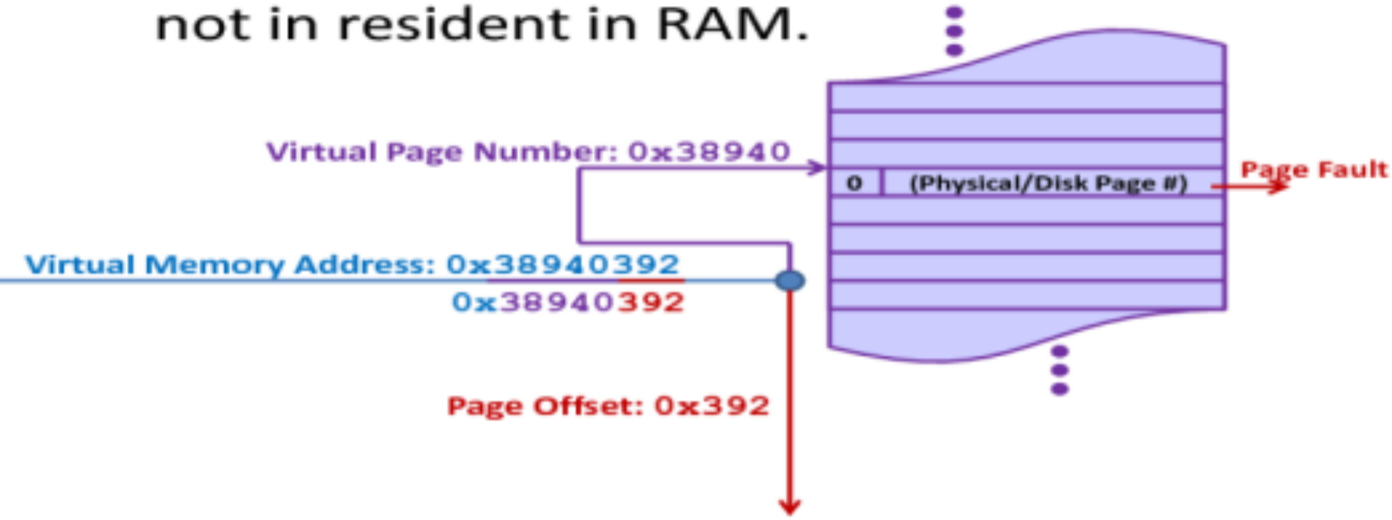
Page Tables So Far

- Each PTE consists of, in part:
  - **Resident Bit**: Is it in RAM or on disk?
  - **Physical Page Number**: Where is it located in RAM or on disk?

- When a page needs to be evicted from RAM (to disk) for another page to be loaded, there are five algorithms:
  - Optimal, FIFO, LRU, LFU, and MRU

Page Fault

- The term **Page Fault** describes the event when a virtual memory address is accessed and is not in resident in RAM.



Page Fault

- When a Page Fault occurs:
  - Check if there is a free page of memory in RAM.
    - If so, load the data to the empty page in RAM.
    - If not, invoke a page replacement algorithm.
      - FIFO, LRU, LFU, MRU, …
      - What does x86 processers use?

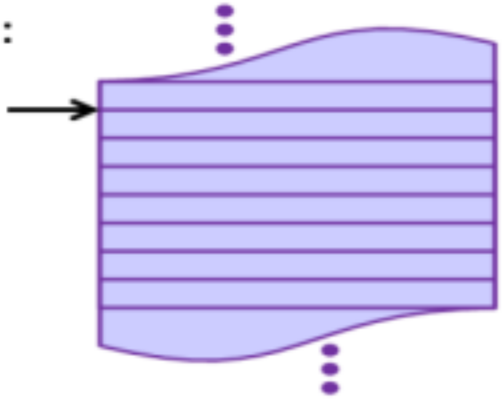
Reference Bit

- A second bit present in modern page tables is a **Reference Bit**.
  - **1**: The page was recently referenced.
  - **0**: The page has not been recently referenced.

- Every time the page is accessed (read/write), the reference bit is set to **1**.

Using the Reference Bit

- When a page needs to be evicted, the page table is scanned.
  - If the page is in RAM (resident):
    - If Ref=1, set it Ref=0.
    - If Ref=0, evict page.
  - Store the pointer to continue the scan at the same position next eviction cycle.



Reference Bit

- The Reference Bit implements a **LRU-like** algorithm with **only 1 bit** of storage /PTE.
  - Used in x86 processors.

- Other algorithms exist for determining page evictions.
  - More bits allow for increasingly complex functionality. (FIFO, LRU, MRU, LRU, etc.)

Evicting Pages: Slow?

- When a page is evicted, the data has to be written to the hard disk.
  - Much slower than RAM
  - Can this be optimized?

Dirty Bit

- Each PTE contains a bit to denote if the page has been written to since it was loaded.
  - **1**: Data is "dirty", has been written.
  - **0**: Data is "clean", same as when it was loaded.

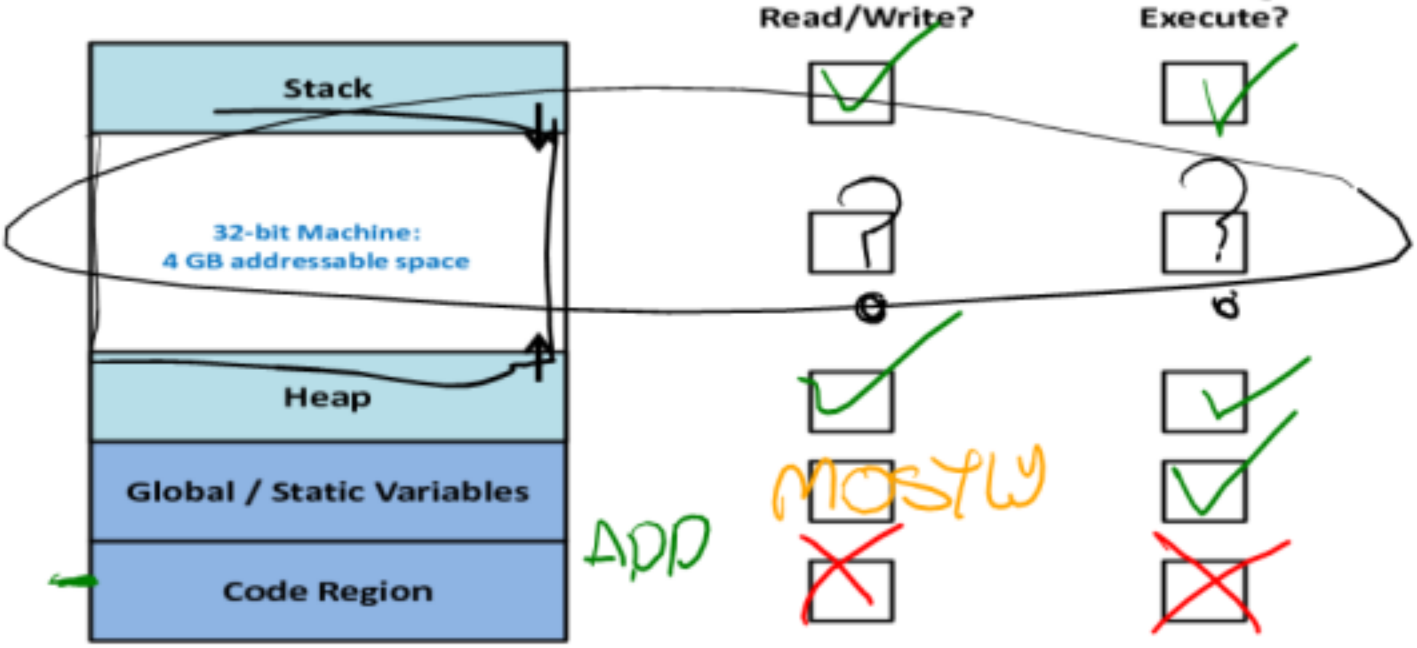
  - Implementation is done in the OS, not hardware.

Protection Bits

- Each PTE also contains bits to protect regions of memory.
  - Read/Write Bit
    - **1**: Enable both reading and writing to the memory.
    - **0**: Enable only reading to the memory.
  - No Execute (NX) Bit
    - **1**: Prevent the memory page's data from being executed.
    - **0**: Allow execution of the memory page's data.

Permission Bits

(NX)



	Read/Write?	Non- Execute?
Stack	☑	☑
32-bit Machine: 4 GB addressable space	☐ ? 0	☐ ? 0
Heap	☑	☑
Global / Static Variables	☐ MOSTLY	☑
Code Region	☒	☒

ADD

Other Bits

- The bits discussed so-far are common across every modern page table implementation:
  - Resident Bit
  - Eviction Bit(s)
    - In x86: Reference Bit
  - Dirty Bit
  - Read/Write Bit
  - NX Bit

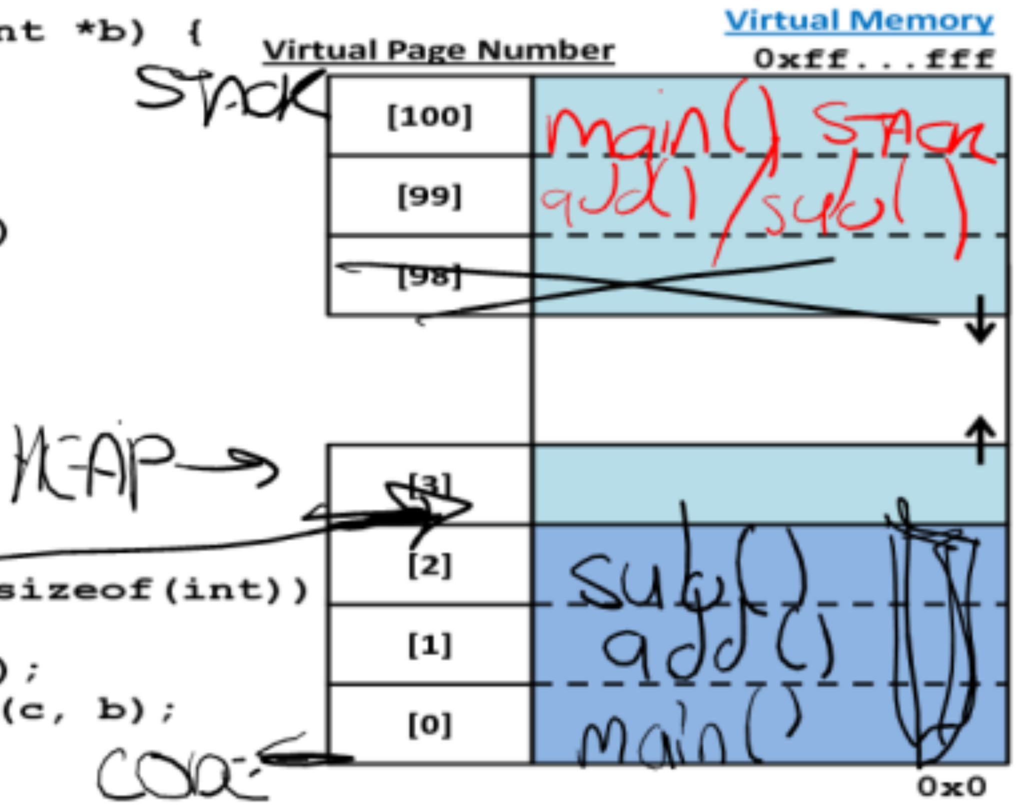
Other Bits

- Other bits are present on PTEs for various purposes:
  - Optimizations
  - Caching
  - Variable-sized Pages
  - Additional Permissions/Protections
  - …

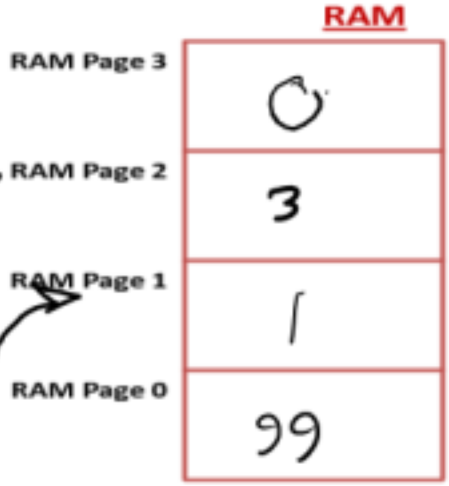
Putting it All Together…

- Lets assume we have another simple system…
  - Size of a page:
    - Enough to store one stack frame **OR**
    - Enough to store one program’s function **OR**
    - Enough to store a small heap

```
int subtract(int a, int *b) {
      int c = a - *b;
      return c;
}

int add(int a, int *b)
{
      int c = a + *b;
      return c;
}

void main()
{
      int a = 4;
      int *b = malloc(sizeof(int))
      *b = 7;
      int c = add(a, b);
      int d = subtract(c, b);
}
```

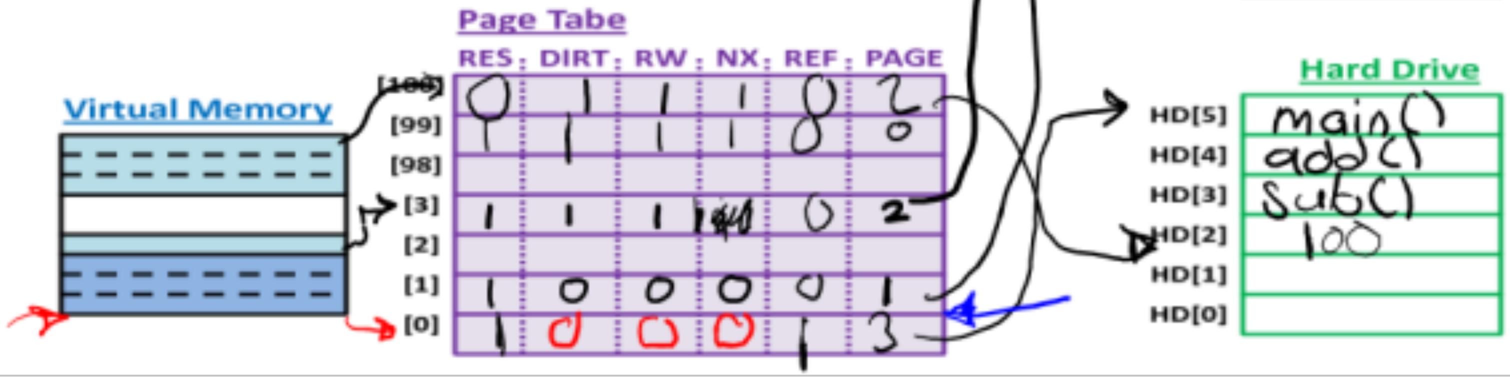


```
int subtract(int a, int *b) {
      int c = a - *b;
      return c;
}

int add(int a, int *b) {
      int c = a + *b;
      return c;
}

void main() {
      int a = 4;
      int *b = malloc(sizeof(int));
      *b = 7
      int c = add(a, b);
      int d = subtract(c, b);
}
```



RAM

RAM Page 3	0
RAM Page 2	3
RAM Page 1	1
RAM Page 0	99



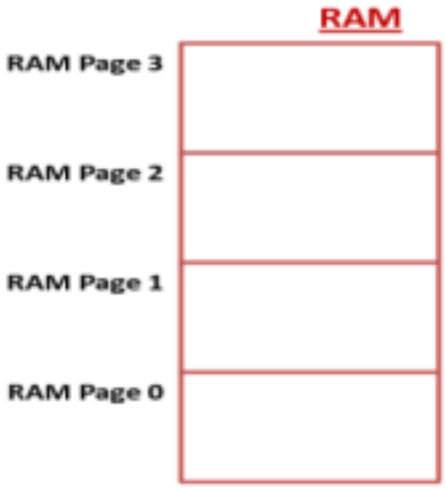
Page Tabe

	RES	DIRT	RW	NX	REF	PAGE
[100]	0	1	1	1	0	2
[99]	1	1	1	1	0	0
[98]						
[3]	1	1	1	[illegible]	0	2
[2]						
[1]	1	0	0	0	0	1
[0]	1	0	0	0	1	3

Hard Drive

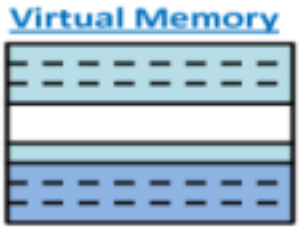
HD[5]	main()
HD[4]	add()
HD[3]	sub()
HD[2]	100
HD[1]	
HD[0]	

```
int subtract(int a, int *b) {
      int c = a - *b;
      return c;
}

int add(int a, int *b) {
      int c = a + *b;
      return c;
}

void main() {
      int a = 4;
      int *b = malloc(sizeof(int));
      *b = 7;
      int c = add(a, b);
      int d = subtract(c, b);
}
```



Page Tabe

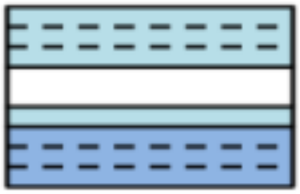
	RES	DIRT	RW	NX	REF	PAGE
[100]						
[99]						
[98]						
[3]						
[2]						
[1]						
[0]						



```
int subtract(int a, int *b) {
      int c = a - *b;
      return c;
}

int add(int a, int *b) {
      int c = a + *b;
      return c;
}

void main() {
      int a = 4;
      int *b = malloc(sizeof(int));
      *b = 7;
      int c = add(a, b);
      int d = subtract(c, b);
}
```

RAM

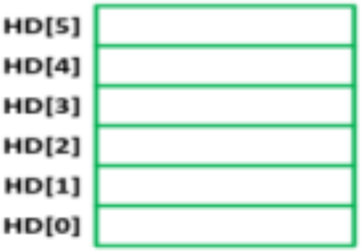
Virtual Memory



Page Tabe

	RES	DIRT	RW	NX	REF	PAGE
[100]						
[99]						
[98]						
[3]						
[2]						
[1]						
[0]						

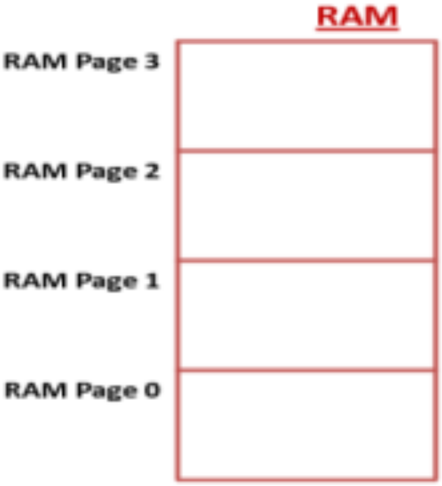
Hard Drive



Multi-Level Page Tables!

CS 241

```
int subtract(int a, int *b) {
      int c = a - *b;
      return c;
}

int add(int a, int *b) {
      int c = a + *b;
      return c;
}

void main() {
      char *b = malloc(sizeof(int));
      *(b + 1000) = 9;
}
```



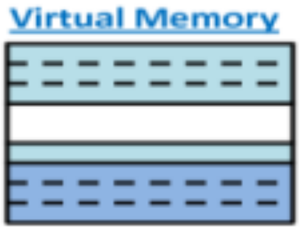
RAM

RAM Page 3

RAM Page 2

RAM Page 1

RAM Page 0



Virtual Memory

Page Tabe

	RES	DIRT	RW	NX	REF	PAGE
[100]						
[99]						
[98]						0
[3]						
[2]						
[1]						
[0]						

Hard Drive

HD[5]

HD[4]

HD[3]

HD[2]

HD[1]

HD[0]

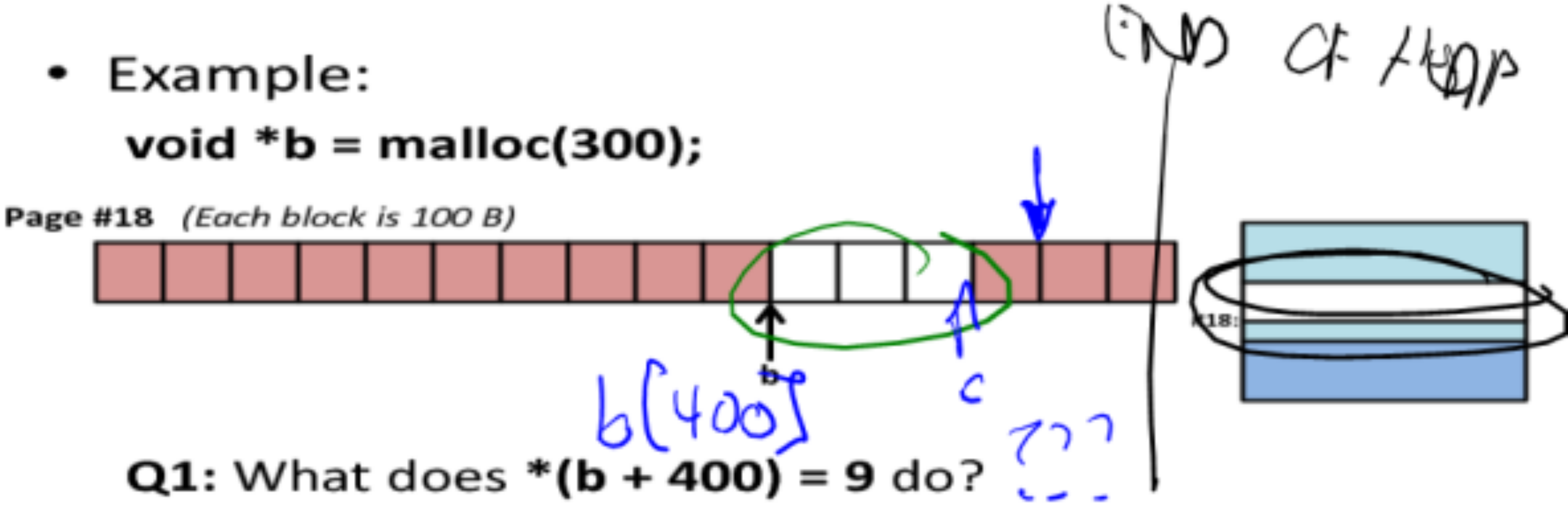
Segmentation Faults

- A “Seg Fault” occurs when an access is made to a virtual memory address that cannot be resolved.

Segmentation Faults

- Example:

  void *b = malloc(300);



Q1: What does ***(b + 400) = 9** do?

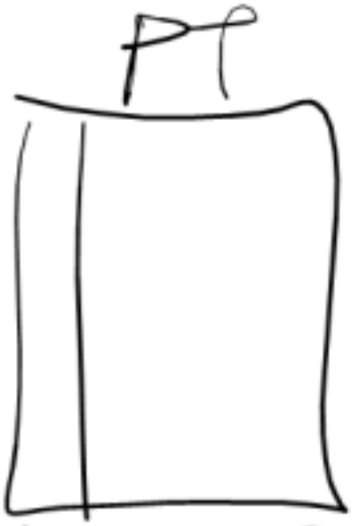
Q2: What does ***(b + 900) = 9** do?

x86 Page Table

0x12345|678

\# | OFFSET

- In x86:
  - Pages are 4 KB in size
  - Virtual Addresses are 32-bits
  - Each PTE is 4 B in size

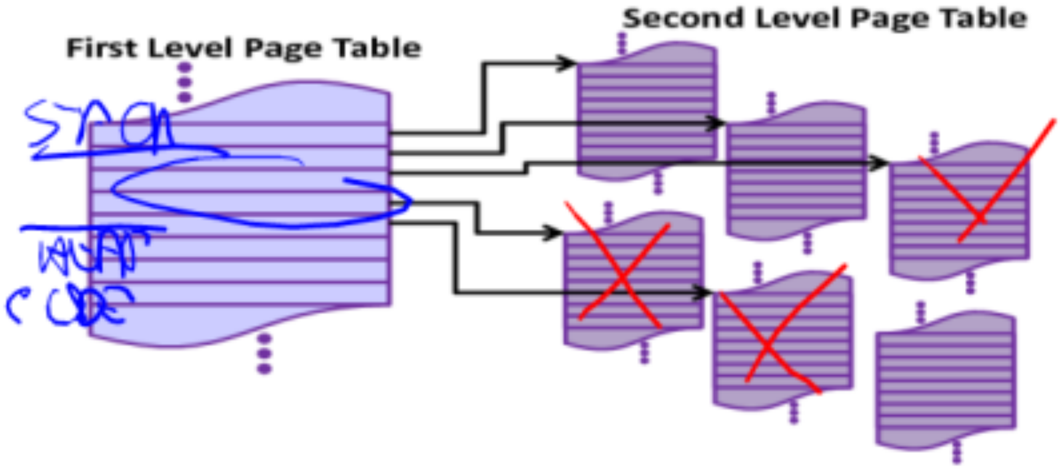


- How large is the Page Table for each process?

$2^{20} * 4 B = 4 MB$

Multi-Level Page Table

- **Solution**: Create multiple levels of tables to look up a physical memory address.



Multi-Level Page Table

- **Advantage:** SAVES SPACE

- **Disadvantage:** SLOW
  - 2 ACCESSES / memory ACCESS

Multi-Level Page Tables

- Each virtual address can now be divided into (n+1) different pieces for an (n) level page table.
  - **Example:** Two Level Page Table:
    - First Level Page Number
    - Second Level Page Number
    - Page Offset

- Given
  - 32-bit Virtual Addresses
  - 4 KB Pages = 12 BITS
  - 12-bit First Level Page Table Number
- What are the components of the address:

0x48503423

12 | 8 | 12

1ST #

OFFSET

- Given
  - 32-bit Virtual Addresses
  - 64 KB Pages = 16
  - 8-bit First Level Page Table Number

0 – F

16 = 2^4

- What are the components of the address:

0x48503423

8 | 8 | 16 BIT

OFFSET

1ST

- Given
  - 32-bit Virtual Addresses
  - 4 KB Pages
  - 4 B page table entries

- How many PTEs fit into one page?

$$\frac{2^{12}\ B}{2^{2}\ B} = \boxed{2^{10}}$$

Multi-Level Page Tables in x86

- In x86, a two-level page table is used.
  - 10-bit Address for the First Level Page Table
  - 10-bit Address for the Second Level Page Table
  - 12-bit Address for the Page Offset

- **Result**:
  - Every single page table fits into one page
  - When a new process is context switched in, only one page needs to initially be loaded for the page table

Review of Memory

- Every process has its own virtual memory address space (0x0 – 0xff…fff).

- Inside that virtual memory space, identify four key regions of memory:
  –
  –
  –
  –

Review of Memory

- To a process, a heap is one contiguous chunk of memory.
  - As memory is allocated and free'd, holes develop in the contiguous chunk of memory.
  - Three strategies to manage this memory space:
    -
    -
    -

Review of Memory

- At a system level, the virtual memory for each process must be mapped to physical storage.

- Two key methods:
  - –
  - –

Review of Memory

- To implement paging, we use a page table made up of page table entries. Key information contained in each PTE includes:
  -
  -
  -
  -
  -
  -

Review of Memory

- When the system runs out of available RAM to store data, pages that likely won't be accessed in the near future are paged-out.
    - Five Strategies:
        -
        -
        -
        -
        -

Review of Memory

- The page table itself is a large data structure. Modern systems break up this page table into multiple levels.
  - **Key Idea**: Identify the number of bits required for every step in memory address translation.
  - Understand the address translation process.