Memory Management

CS 452

Part 1

Abstracting a CPU

- Have discussed at length the concept of a process
- Process = an abstraction of a CPU
  - Allows processes to not have to worry about the details of sharing the CPU hardware, can act as if they “own” it
- How do we handle ***memory***?

A Simple Scenario

- Computer has exactly 16 bytes of memory
- Addresses 0x0 through 0xf
- Let’s consider how this would work for a few simple scenarios

Scenario: One Program

Program A

```
i = 0
while i <= 0x7:
  load r1 mem(i)
  r1++
  store r1 mem(i)
  i++
load temp mem(0x5)
print(temp)
```

Address	Value
0x0	
0x1	
0x2	
0x3	
0x4	
0x5	
0x6	
0x7	
0x8	
0x9	
0xa	
0xb	
0xc	
0xd	
0xe	
0xf	

*** Note: This is an over-simplification - we would need memory for the instructions, vars, etc too*

Scenario: Two Programs

Program A

```
i = 0
while i <= 0x7:
  load r1 mem(i)
  r1++
  store r1 mem(i)
  i++
load temp mem(0x5)
print(temp)
```

Program B

```
i = 0
while i <= 0x7:
  load r1 mem(i)
  r1 = r1 * 2
  store r1 mem(i)
  i++
load temp mem(0x4)
print(temp)
```

Address	Value
0x0	
0x1	
0x2	
0x3	
0x4	
0x5	
0x6	
0x7	
0x8	
0x9	
0xa	
0xb	
0xc	
0xd	
0xe	
0xf	

*** Note: This is an over-simplification - we would need memory for the instructions, vars, etc too*

Scenario: Two Programs

Program A

```
i = 0
while i <= 0x7:
  load r1 mem(i)
  r1++
  store r1 mem(i)
  i++
load temp mem(0x5)
print(temp)
```

Program B

```
i = 0
while i <= 0x7:
  load r1 mem(i)
  r1 = r1 * 2
  store r1 mem(i)
  i++
load temp mem(0x4)
print(temp)
```

What happens if a context switch from A to B happens half-way through execution?

Address	Value
0x0	
0x1	
0x2	
0x3	
0x4	
0x5	
0x6	
0x7	
0x8	
0x9	
0xa	
0xb	
0xc	
0xd	
0xe	
0xf	

Scenario: Two Programs

Program A

```
i = 0
while i <= 0x7:
   load r1 mem(i)
   r1++
   store r1 mem(i)
   i++
load temp mem(0x5)
print(temp)
```

Program B

```
i = 0
while i <= 0x7:
   load r1 mem(i)
   r1 = r1 * 2
   store r1 mem(i)
   i++
load temp mem(0x4)
print(temp)
```

What are possible solutions?

Address	Value
0x0	
0x1	
0x2	
0x3	
0x4	
0x5	
0x6	
0x7	
0x8	
0x9	
0xa	
0xb	
0xc	
0xd	
0xe	
0xf	

Scenario: Two Programs

Program A

```
i = 0
while i <= 0x7:
   load r1 mem(i)
   r1++
   store r1 mem(i)
   i++
load temp mem(0x5)
print(temp)
```

Program B

```
i = 0
while i <= 0x7:
   load r1 mem(i)
   r1 = r1 * 2
   store r1 mem(i)
   i++
load temp mem(0x4)
print(temp)
```

Solution 1: Each program gets full memory, save to disk when CSed away from, load when CSed back to

Address	Value
0x0	
0x1	
0x2	
0x3	
0x4	
0x5	
0x6	
0x7	
0x8	
0x9	
0xa	
0xb	
0xc	
0xd	
0xe	
0xf	

Scenario: Two Programs

Program A

```
i = 0
while i <= 0x7:
  load r1 mem(i)
  r1++
  store r1 mem(i)
  i++
load temp mem(0x5)
print(temp)
```

Program B

```
i = 0
while i <= 0x7:
  load r1 mem(i)
  r1 = r1 * 2
  store r1 mem(i)
  i++
load temp mem(0x4)
print(temp)
```

Solution 2: Programs only get as much memory as they need. Have a limited range. Problems??

Address	Value
0x0	
0x1	
0x2	
0x3	
0x4	
0x5	
0x6	
0x7	
0x8	
0x9	
0xa	
0xb	
0xc	
0xd	
0xe	
0xf	

Scenario: Two Programs

Program A

```
i = 0
while i <= 0x7:
  load r1 mem(i)
  r1++
  store r1 mem(i)
  i++
```

Program B

```
i = 0
while i <= 0x7:
  load r1 mem(i)
  r1 = r1 * 2
  store r1 mem(i)
  i++
```

Problems:

(A) How to know address range in advance?

(B) What is there isn't enough space?

Address	Value
0x0	
0x1	
0x2	
0x3	
0x4	
0x5	
0x6	
0x7	
0x8	
0x9	
0xa	
0xb	
0xc	
0xd	
0xe	
0xf	

Scenario: Two Programs

Program A

```
i = 0
while i <= 0x7:
  load r1 mem(i)
  r1++
  store r1 mem(i)
  i++
```

BASE = 0x0

Program B

```
i = 0
while i <= 0x7:
  load r1 mem(i)
  r1 = r1 * 2
  store r1 mem(i)
  i++
```

BASE = 0x8

Address	Value
0x0	
0x1	
0x2	
0x3	
0x4	
0x5	
0x6	
0x7	
0x8	
0x9	
0xa	
0xb	
0xc	
0xd	
0xe	
0xf	

Solution for A:
Each is written to assume it starts at 0x0, is given an offset address

Dynamic Relocation

- This technique is known as **dynamic relocation**
- Either when program loaded or as instructions execute, dynamically change the addresses if need be
- **BASE** Register: The offset for all addresses the program uses
- **LIMIT** Register: Represents the max address
  - Combined gives us an **Address Space!**
  - Each program can use any address it wants between 0x0 and **LIMIT**. Kernel / CPU handles the offset.

Scenario: Three Programs

Program A

```
i = 0
while i <= 0x4:
  load r1 mem(i)
  r1++
  store r1 mem(i)
  i++
```

BASE = 0x0

Program B

```
i = 0x0
while i <= 0x7:
  load r1 mem(i)
  r1 = r1 * 2
  store r1 mem(i)
  i++
```

BASE = 0x5

Program C

```
c = 25
store c mem(0x0)
store c mem(0x1)
store c mem(0x2)
store c mem(0x3)
```

BASE = 0xd

Address	Value
0x0	
0x1	
0x2	
0x3	
0x4	
0x5	
0x6	
0x7	
0x8	
0x9	
0xa	
0xb	
0xc	
0xd	
0xe	
0xf	

Scenario: Three Programs

Program A

```
i = 0
while i <= 0x4:
   load r1 mem(i)
   r1++
   store r1 mem(i)
   i++
```

BASE = 0x0

Program B

```
i = 0x0
while i <= 0x7:
   load r1 mem(i)
   r1 = r1 * 2
   store r1 mem(i)
   i++
```

BASE = 0x5

Program C

```
c = 25
store c mem(0x0)
store c mem(0x1)
store c mem(0x2)
store c mem(0x3)
```

BASE = 0xd

Address	Value
0x0	
0x1	
0x2	
0x3	
0x4	
0x5	
0x6	
0x7	
0x8	
0x9	
0xa	
0xb	
0xc	
0xd	
0xe	
0xf	

What is the problem?

Scenario: Three Programs

Program A

```
i = 0
while i <= 0x4:
   load r1 mem(i)
   r1++
   store r1 mem(i)
   i++
```

BASE = 0x0

Program B

```
i = 0x0
while i <= 0x7:
   load r1 mem(i)
   r1 = r1 * 2
   store r1 mem(i)
   i++
```

BASE = 0x5

Program C

```
c = 25
store c mem(0x0)
store c mem(0x1)
store c mem(0x2)
store c mem(0x3)
```

BASE = 0xd

Address	Value
0x0	
0x1	
0x2	
0x3	
0x4	
0x5	
0x6	
0x7	
0x8	
0x9	
0xa	
0xb	
0xc	
0xd	
0xe	
0xf	

Not enough space!
We are one byte of memory short.
Program C either needs to wait (could starve) or could copy to memory

A Different Scenario

- Don’t change anything, except:
  - Computer has more memory 64kb bytes of memory
  - Addresses 0x0 through 0xffff
- Now we have PLENTY of memory to run tons of programs! (😂)
- But…. there still is a problem

Scenario: Plenty of Memory

Program A

```
i = 0
while i <= 0x4:
  load r1 mem(i)
  r1++
  store r1 mem(i)
  i++
```

Address	Value
0x0	
0x1	
0x2	
0x3	
0x4	
0x5	
0x6	
0x7	
0x8	
0x9	
0xa	
0xb	
0xc	
0xd	
0xe	
0xf	

…. plenty more ….

Scenario: Plenty of Memory

Program A

```
i = 0
while i <= 0x4:
  load r1 mem(i)
  r1++
  store r1 mem(i)
  i++
```

BASE = 0x0

Address	Value
0x0	
0x1	
0x2	
0x3	
0x4	
0x5	
0x6	
0x7	
0x8	
0x9	
0xa	
0xb	
0xc	
0xd	
0xe	
0xf	

…. plenty more ….

Scenario: Plenty of Memory

Program A

```
i = 0
while i <= 0x4:
  load r1 mem(i)
  r1++
  store r1 mem(i)
  i++
```

Program B

```
i = 0x0
while i <= 0x8:
  load r1 mem(i)
  r1 = r1 * 2
  store r1 mem(i)
  i++
```

BASE = 0x0

Address	Value
0x0	
0x1	
0x2	
0x3	
0x4	
0x5	
0x6	
0x7	
0x8	
0x9	
0xa	
0xb	
0xc	
0xd	
0xe	
0xf	

…. plenty more ….

Scenario: Plenty of Memory

Program A

```
i = 0
while i <= 0x4:
  load r1 mem(i)
  r1++
  store r1 mem(i)
  i++
```

BASE = 0x0

Program B

```
i = 0x0
while i <= 0x7:
  load r1 mem(i)
  r1 = r1 * 2
  store r1 mem(i)
  i++
```

BASE = 0x5

Address	Value
0x0	
0x1	
0x2	
0x3	
0x4	
0x5	
0x6	
0x7	
0x8	
0x9	
0xa	
0xb	
0xc	
0xd	
0xe	
0xf	

…. plenty more ….

Scenario: Plenty of Memory

Program A

```
i = 0
while i <= 0x4:
   load r1 mem(i)
   r1++
   store r1 mem(i)
   i++
malloc(input())
```

BASE = 0x0

Program B

```
i = 0x0
while i <= 0x7:
   load r1 mem(i)
   r1 = r1 * 2
   store r1 mem(i)
   i++
```

BASE = 0x5

Process A needs more memory, what to do?

Address	Value
0x0	
0x1	
0x2	
0x3	
0x4	
0x5	
0x6	
0x7	
0x8	
0x9	
0xa	
0xb	
0xc	
0xd	
0xe	
0xf	

…. plenty more ….

Scenario: Plenty of Memory

Program A

```
i = 0
while i <= 0x4:
   load r1 mem(i)
   r1++
   store r1 mem(i)
   i++
malloc(input())
```

BASE = 0x0

Program B

```
i = 0x0
while i <= 0x7:
   load r1 mem(i)
   r1 = r1 * 2
   store r1 mem(i)
   i++
```

BASE = 0x5

Process A needs more memory, what to do?

- Move it to a new location?

- Shift process B down?

Address	Value
0x0	
0x1	
0x2	
0x3	
0x4	
0x5	
0x6	
0x7	
0x8	
0x9	
0xa	
0xb	
0xc	
0xd	
0xe	
0xf	

…. plenty more ….

Not Satisfying

- Each of the proposed solutions has one or more weaknesses
- Need a mechanism that gives each process its own **abstract memory space**, while still being relatively efficient.
- An Answer….

Virtual Memory with Page Tables

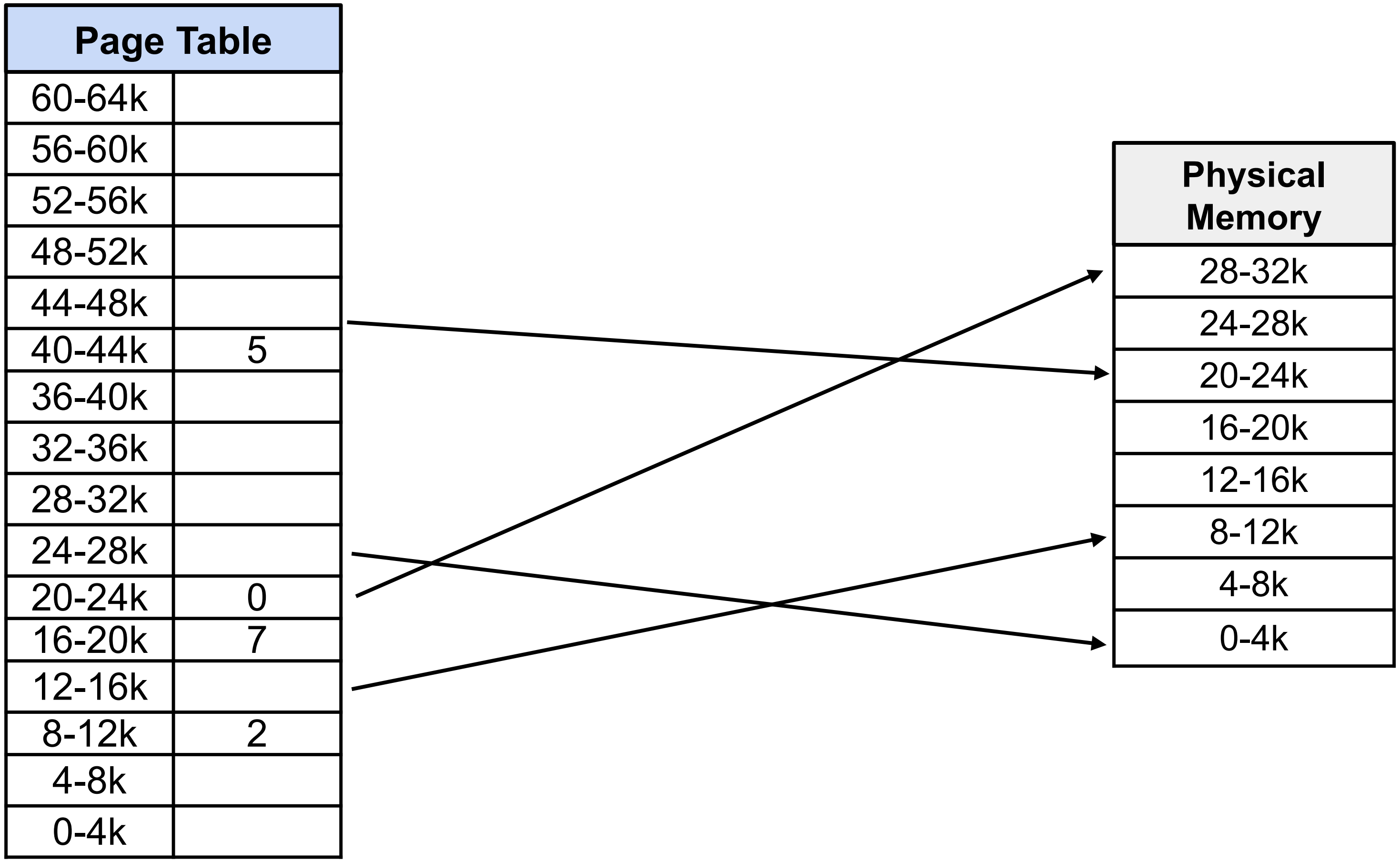
- **Virtual Memory** is the technique used in many modern OSs
- Each process gets to act as if it owns a large memory space completely to itself.
  - EG, on a 64-bit system, addresses `0x000000000000` to `0xffffffffffff`
- A combination of the OS kernel and the hardware manage the mapping between each process's **virtual address space** and the **physical address space** of the RAM.

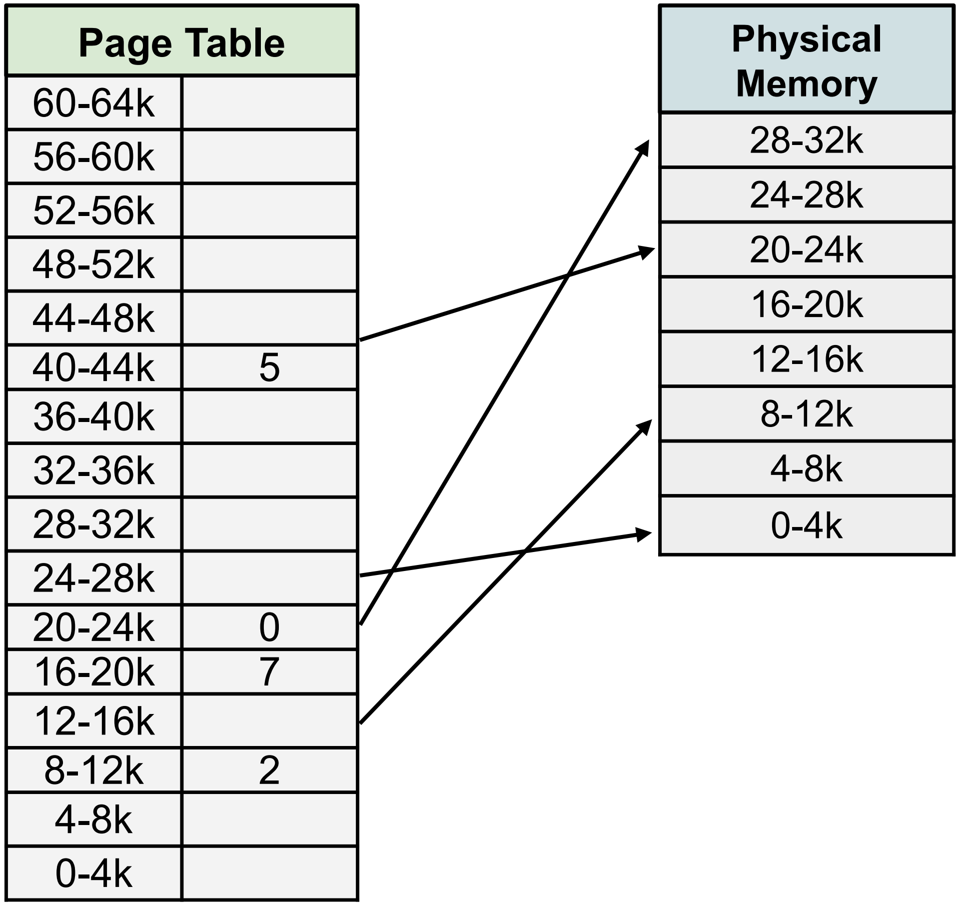
NOTE: No need for BASE / LIMIT anymore!

Virtual Memory with Page Tables

- The **page table** is a key structure to allow for this to happen with reasonable efficiency
- The OS keeps a **page table** for each process
- Maps virtual addresses to physical ones
- Divides virtual memory spaces into pages (for example, 4k chunks of bytes)

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	X
24-28k	1
20-24k	X
16-20k	X
12-16k	5
8-12k	X
4-8k	X
0-4k	0


Page Table
60-64k
56-60k
52-56k
48-52k
44-48k
40-44k
5
36-40k
32-36k
28-32k
24-28k
20-24k
0
16-20k
7
12-16k
8-12k
2
4-8k
0-4k
Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k



Page Table
60-64k
56-60k
52-56k
48-52k
44-48k
40-44k
5
36-40k
32-36k
28-32k
24-28k
20-24k
0
16-20k
7
12-16k
8-12k
2
4-8k
0-4k
Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

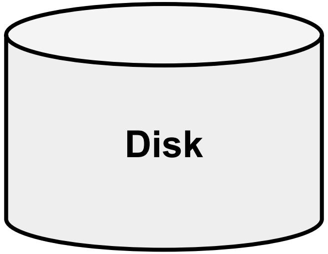
New Scenario

- Single-core computer
- Has 32k of physical memory
- Each process gets a **virtual address space** of 64k
- Page size is 4k
  - 8 physical page slots, 16 virtual pages per process

Program A

```
i = 0
while i <= 54k:
  load r1 mem(i)
  r1++
  store r1 mem(i)
  i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

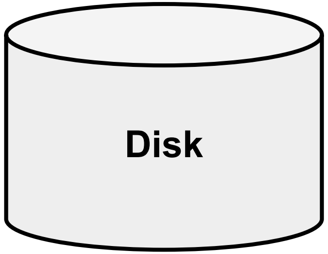


Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

```
i = 0
while i <= 54k:
  load r1 mem(i)
  r1++
  store r1 mem(i)
  i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	0



Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

```
i = 0
while i <= 54k:
  load r1 mem(i)
  r1++
  store r1 mem(i)
  i++
```

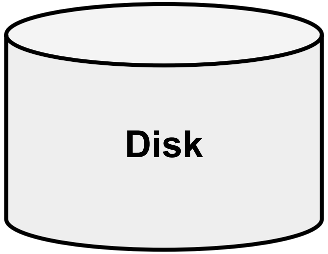
Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	1
0-4k	0

Disk

Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

```
i = 0
while i <= 54k:
  load r1 mem(i)
  r1++
  store r1 mem(i)
  i++
```

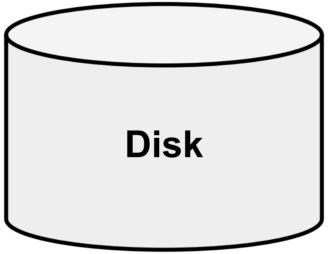


Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	2
4-8k	1
0-4k	0

Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

```
i = 0
while i <= 54k:
  load r1 mem(i)
  r1++
  store r1 mem(i)
  i++
```



Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	7
24-28k	6
20-24k	5
16-20k	4
12-16k	3
8-12k	2
4-8k	1
0-4k	0

....

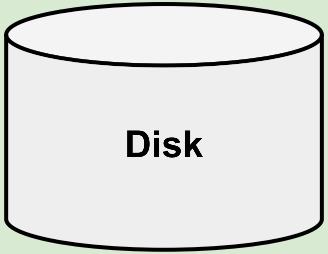
Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

```
i = 0
while i <= 54k:
    load r1 mem(i)
    r1++
    store r1 mem(i)
    i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	7
24-28k	6
20-24k	5
16-20k	4
12-16k	3
8-12k	2
4-8k	1
0-4k	0

What happens next?



. . . .

Physical Memory

28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

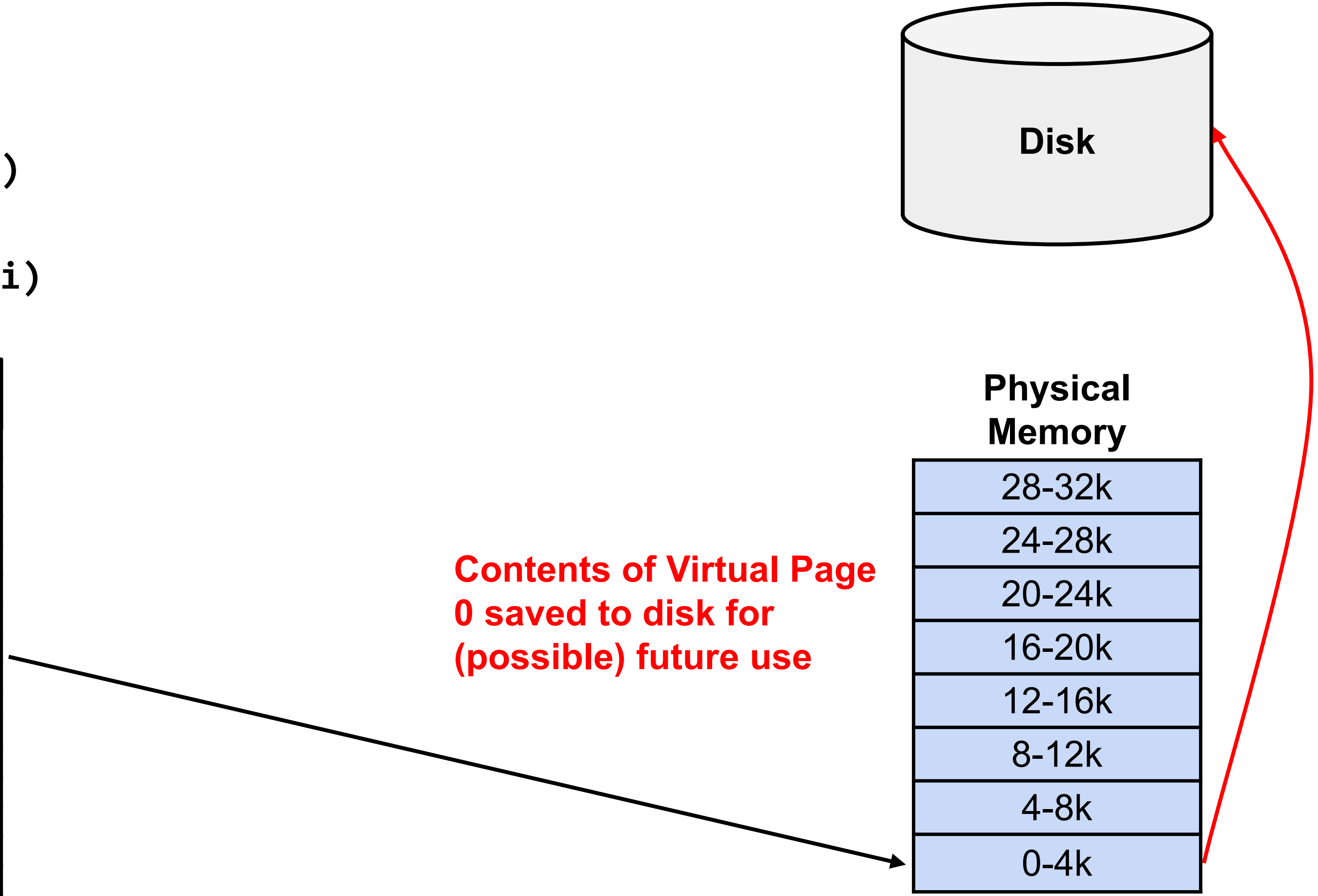
Storing Pages on Disk

- Sometimes, need to store pages on disk (hard drive, SSD, etc)
  - Virtual memory space often is larger than physical memory
  - Even if that was not the case, multiple processes
- These are stored / loaded into a **swap partition** or **swap file**
  - When page evicted, save if need be
  - When page table has miss, load from swap file
- See **`$ cat /proc/PID/maps`**

Program A

```
i = 0
while i <= 54k:
   load r1 mem(i)
   r1++
   store r1 mem(i)
   i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	0
28-32k	7
24-28k	6
20-24k	5
16-20k	4
12-16k	3
8-12k	2
4-8k	1
0-4k	X



Notice: This is now invalid!

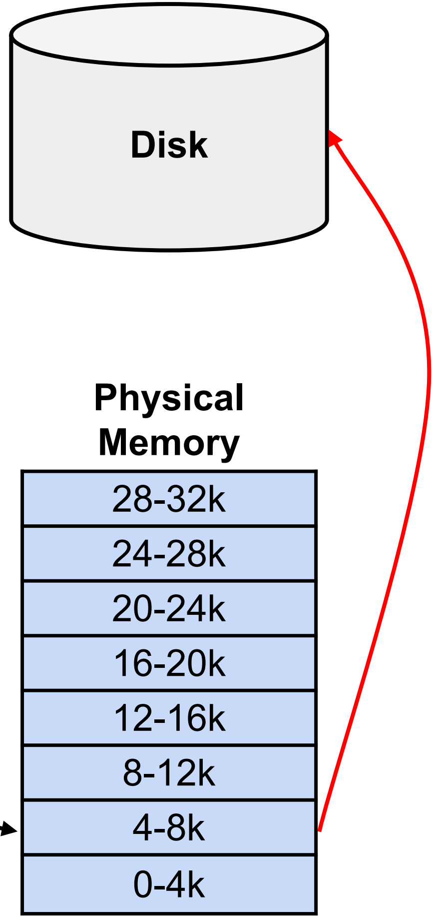
Program A

```
i = 0
while i <= 54k:
    load r1 mem(i)
    r1++
    store r1 mem(i)
    i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	1
32-36k	0
28-32k	7
24-28k	6
20-24k	5
16-20k	4
12-16k	3
8-12k	2
4-8k	X
0-4k	X

Contents of Virtual Page 1 saved to disk for (possible) future use)

Notice: This is now invalid!



Program A

```
i = 0
while i <= 54k:
   load r1 mem(i)
   r1++
   store r1 mem(i)
   i++
```

Context Switch!

Program B

```
i = 20k
while i <= 40k:
   load r1 mem(i)
   r1 = r1 * 2
   store r1 mem(i)
   i++
```

Disk

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	1
32-36k	0
28-32k	7
24-28k	6
20-24k	5
16-20k	4
12-16k	3
8-12k	2
4-8k	X
0-4k	X

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

```
i = 0
while i <= 54k:
   load r1 mem(i)
   r1++
   store r1 mem(i)
   i++
```

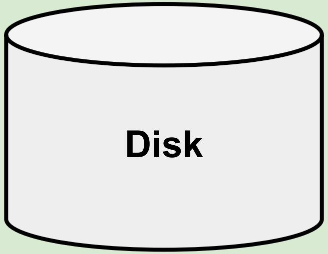
Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	1
32-36k	0
28-32k	7
24-28k	6
20-24k	5
16-20k	4
12-16k	3
8-12k	2
4-8k	X
0-4k	X

Context Switch!

Program B

```
i = 20k
while i <= 40k:
   load r1 mem(i)
   r1 = r1 * 2
   store r1 mem(i)
   i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X



How to choose next physical memory location to use?

Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

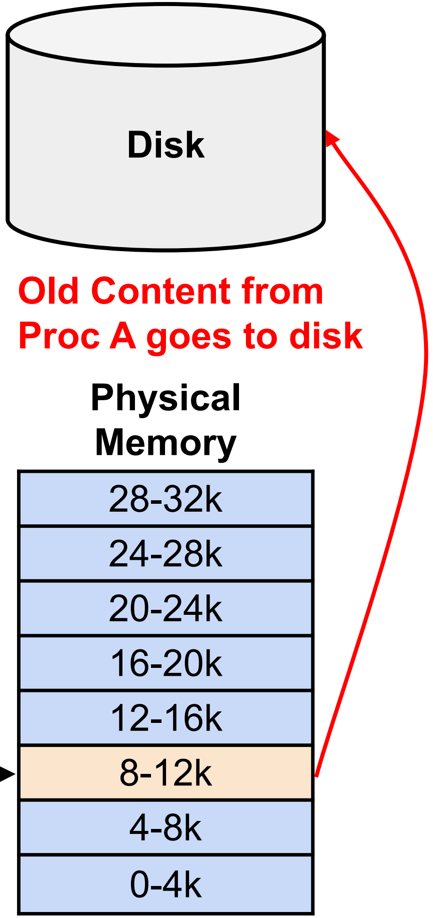
```
i = 0
while i <= 54k:
   load r1 mem(i)
   r1++
   store r1 mem(i)
   i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	1
32-36k	0
28-32k	7
24-28k	6
20-24k	5
16-20k	4
12-16k	3
8-12k	X
4-8k	X
0-4k	X

Program B

```
i = 20k
while i <= 40k:
   load r1 mem(i)
   r1 = r1 * 2
   store r1 mem(i)
   i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	2
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X



Program A

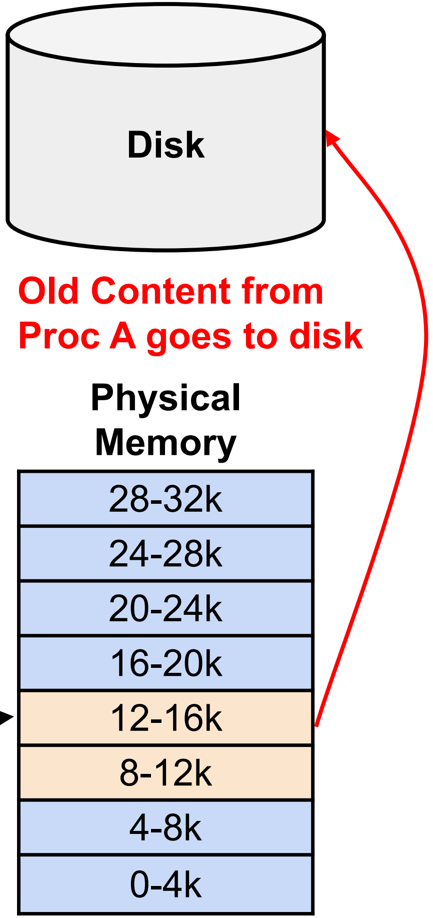
```
i = 0
while i <= 54k:
   load r1 mem(i)
   r1++
   store r1 mem(i)
   i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	1
32-36k	0
28-32k	7
24-28k	6
20-24k	5
16-20k	4
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Program B

```
i = 20k
while i <= 40k:
   load r1 mem(i)
   r1 = r1 * 2
   store r1 mem(i)
   i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	X
24-28k	3
20-24k	2
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X



Program A

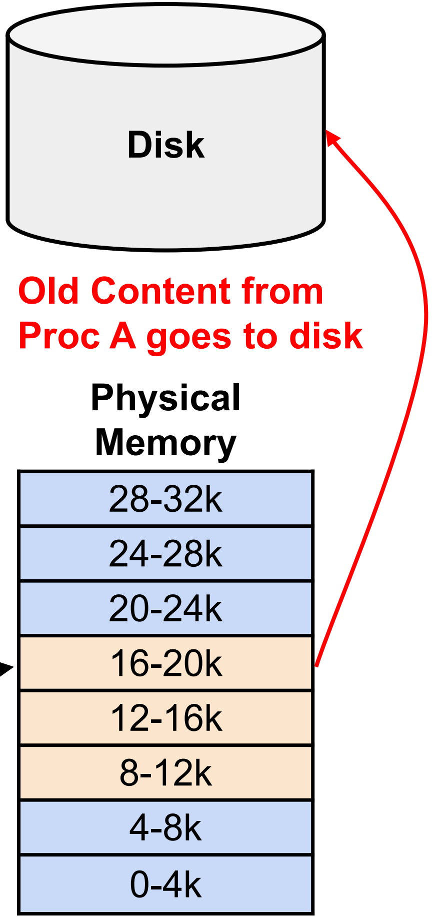
```
i = 0
while i <= 54k:
   load r1 mem(i)
   r1++
   store r1 mem(i)
   i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	1
32-36k	0
28-32k	7
24-28k	6
20-24k	5
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Program B

```
i = 20k
while i <= 40k:
   load r1 mem(i)
   r1 = r1 * 2
   store r1 mem(i)
   i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	4
24-28k	3
20-24k	2
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X



Program A

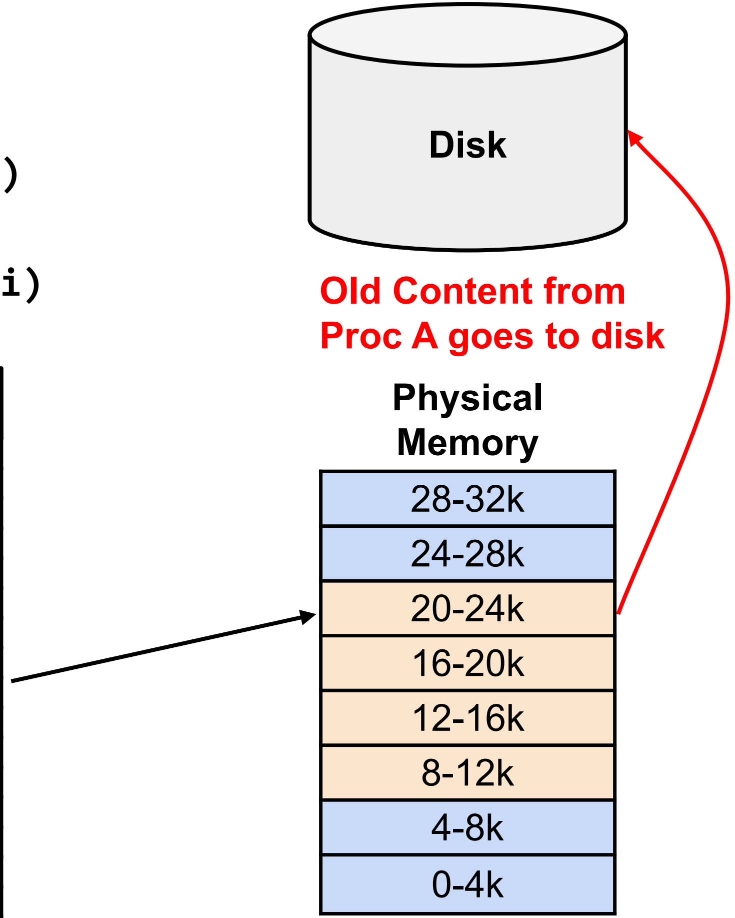
```
i = 0
while i <= 54k:
   load r1 mem(i)
   r1++
   store r1 mem(i)
   i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	1
32-36k	0
28-32k	7
24-28k	6
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Program B

```
i = 20k
while i <= 40k:
   load r1 mem(i)
   r1 = r1 * 2
   store r1 mem(i)
   i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	5
28-32k	4
24-28k	3
20-24k	2
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X



Program A

```
i = 0
while i <= 54k:
   load r1 mem(i)
   r1++
   store r1 mem(i)
   i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	1
32-36k	0
28-32k	7
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Program B

```
i = 20k
while i <= 40k:
   load r1 mem(i)
   r1 = r1 * 2
   store r1 mem(i)
   i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	6
32-36k	5
28-32k	4
24-28k	3
20-24k	2
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Disk

Old Content from Proc A goes to disk

Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

```
i = 0
while i <= 54k:
   load r1 mem(i)
   r1++
   store r1 mem(i)
   i++
```

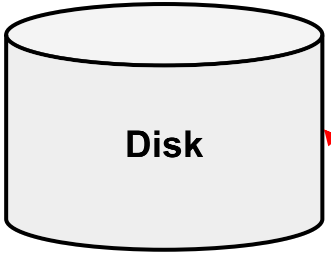
Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	1
32-36k	0
28-32k	7
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Context Switch!

Program B

```
i = 20k
while i <= 40k:
   load r1 mem(i)
   r1 = r1 * 2
   store r1 mem(i)
   i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	6
32-36k	5
28-32k	4
24-28k	3
20-24k	2
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X



Old Content from Proc A goes to disk

Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

```
i = 0
while i <= 54k:
   load r1 mem(i)
   r1++
   store r1 mem(i)
   i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	7
36-40k	1
32-36k	0
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Program B

```
i = 20k
while i <= 40k:
   load r1 mem(i)
   r1 = r1 * 2
   store r1 mem(i)
   i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	6
32-36k	5
28-32k	4
24-28k	3
20-24k	2
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Disk

Old Content from Proc A goes to disk

Physical Memory

28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

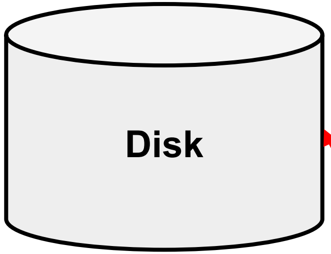
```
i = 0
while i <= 54k:
   load r1 mem(i)
   r1++
   store r1 mem(i)
   i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	0
40-44k	7
36-40k	1
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Program B

```
i = 20k
while i <= 40k:
   load r1 mem(i)
   r1 = r1 * 2
   store r1 mem(i)
   i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	6
32-36k	5
28-32k	4
24-28k	3
20-24k	2
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X



Old Content from Proc A goes to disk

Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

```
i = 0
while i <= 54k:
   load r1 mem(i)
   r1++
   store r1 mem(i)
   i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	1
44-48k	0
40-44k	7
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Program B

```
i = 20k
while i <= 40k:
   load r1 mem(i)
   r1 = r1 * 2
   store r1 mem(i)
   i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	6
32-36k	5
28-32k	4
24-28k	3
20-24k	2
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Disk

Old Content from Proc A goes to disk

Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

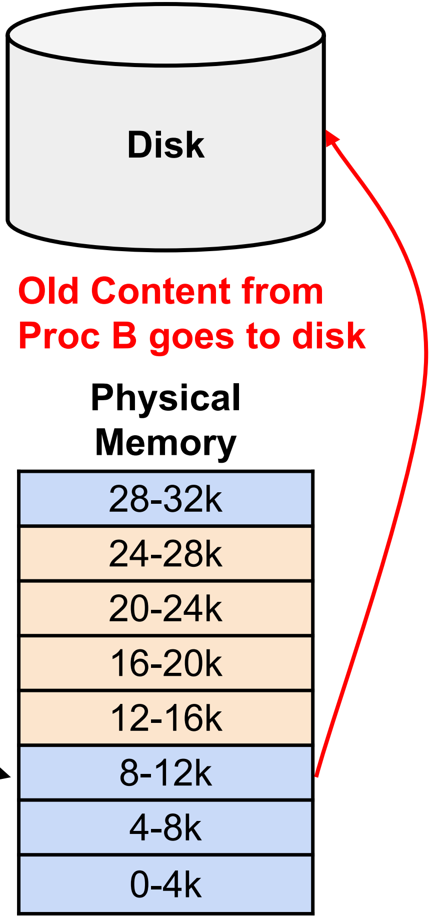
```
i = 0
while i <= 54k:
   load r1 mem(i)
   r1++
   store r1 mem(i)
   i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	2
48-52k	1
44-48k	0
40-44k	7
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Program B

```
i = 20k
while i <= 40k:
   load r1 mem(i)
   r1 = r1 * 2
   store r1 mem(i)
   i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	6
32-36k	5
28-32k	4
24-28k	3
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X



Program A

```
i = 0
while i <= 54k:
. . . .
load r1 0x10
```

Program B

```
i = 20k
while i <= 40k:
   load r1 mem(i)
   r1 = r1 * 2
   store r1 mem(i)
   i++
```

Disk

Page Table	
60-64k	X
56-60k	X
52-56k	2
48-52k	1
44-48k	0
40-44k	7
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	6
32-36k	5
28-32k	4
24-28k	3
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

What happens if this is the next instruction executed?

Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Non-Sequential

- Entries in the page table do not have to be sequential
- Programs can have “sequential” access patterns, but also “random” access patterns as well

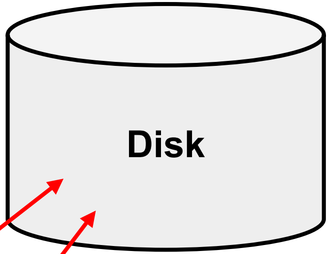
Program A

```
i = 0
while i <= 54k:
. . . .
load r1 0x10
```

Invalid !

Program B

```
i = 20k
while i <= 40k:
   load r1 mem(i)
   r1 = r1 * 2
   store r1 mem(i)
   i++
```



Page Table	
60-64k	X
56-60k	X
52-56k	2
48-52k	1
44-48k	0
40-44k	7
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	6
32-36k	5
28-32k	4
24-28k	3
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Save!

Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

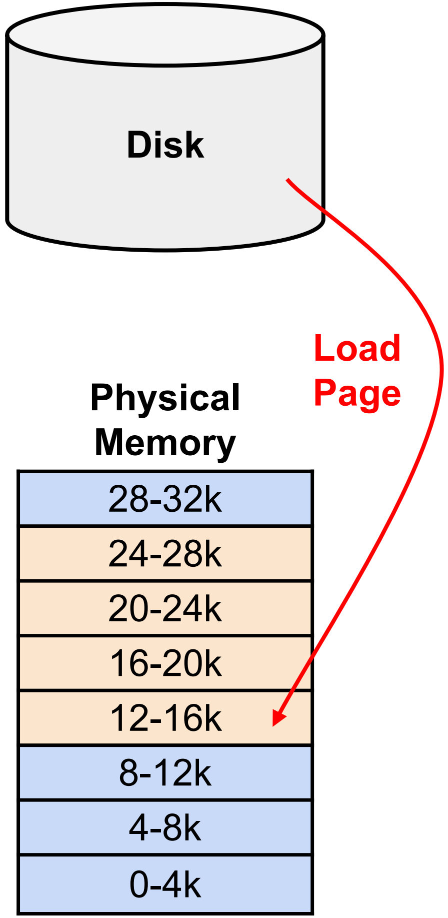
```
i = 0
while i <= 54k:
. . . .
load r1 0x10
```

Page Table	
60-64k	X
56-60k	X
52-56k	2
48-52k	1
44-48k	0
40-44k	7
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Program B

```
i = 20k
while i <= 40k:
    load r1 mem(i)
    r1 = r1 * 2
    store r1 mem(i)
    i++
```

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	6
32-36k	5
28-32k	4
24-28k	3
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

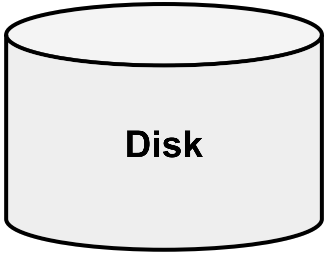


Program A

```
i = 0
while i <= 54k:
. . . .
load r1 0x10
```

Program B

```
i = 20k
while i <= 40k:
  load r1 mem(i)
  r1 = r1 * 2
  store r1 mem(i)
  i++
```



Page Table	
60-64k	X
56-60k	X
52-56k	2
48-52k	1
44-48k	0
40-44k	7
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	3

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	6
32-36k	5
28-32k	4
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

```
i = 0
while i <= 54k:
. . . .
load r1 0x10
```

Context Switch!

Program B

```
i = 20k
while i <= 40k:
. . . .
store r2 0x5208
```

Disk

Page Table (Program A)

60-64k	X
56-60k	X
52-56k	2
48-52k	1
44-48k	0
40-44k	7
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	3

Page Table (Program B)

60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	6
32-36k	5
28-32k	4
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Physical Memory

28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

```
i = 0
while i <= 54k:
. . . .
load r1 0x10
```

Program B

```
store r2 0x5208
i = 20k
while i <= 40k:
. . . .
store r2 0x5208
```

// Dec 21000

Page Table (Program A)

Page Table	
60-64k	X
56-60k	X
52-56k	2
48-52k	1
44-48k	0
40-44k	7
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	3

Page Table (Program B)

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	6
32-36k	5
28-32k	4
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Invalid!

Disk

Save and Load

Physical Memory

Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

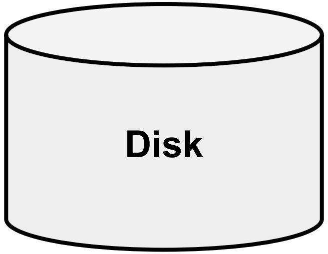
Program A

```
i = 0
while i <= 54k:
. . . .
load r1 0x10
```

Program B

```
i = 20k
while i <= 40k:
. . . .
store r2 0x5208
```

// Dec 21000



Page Table (Program A)

Page Table	
60-64k	X
56-60k	X
52-56k	2
48-52k	1
44-48k	0
40-44k	7
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	3

Page Table (Program B)

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	6
32-36k	5
28-32k	X
24-28k	X
20-24k	4
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Physical Memory

Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Page Size

Is a 4k Page size a good idea?

What happens if we make the pages size larger? 8k? 16k?

What about smaller?

Page Table Lookup

- How does a page table lookup work?
- The incoming address is split up in to two components a **Page Table Index** (PTI) and an **Offset**.
- PTI used to determine slot in page table
- Offset is appended to the value stored in page table to get physical address

Present / Absent bit

Page Table index (Decimal)

Physical Page Number

Page Table index	Page Table	
15	000	0
14	000	0
13	000	0
12	000	0
11	111	1
10	000	0
9	101	1
8	000	0
7	000	0
6	000	0
5	011	1
4	100	1
3	000	1
2	110	1
1	001	1
0	010	1

Incoming Virtual Address

0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0

Outgoing Physical Address

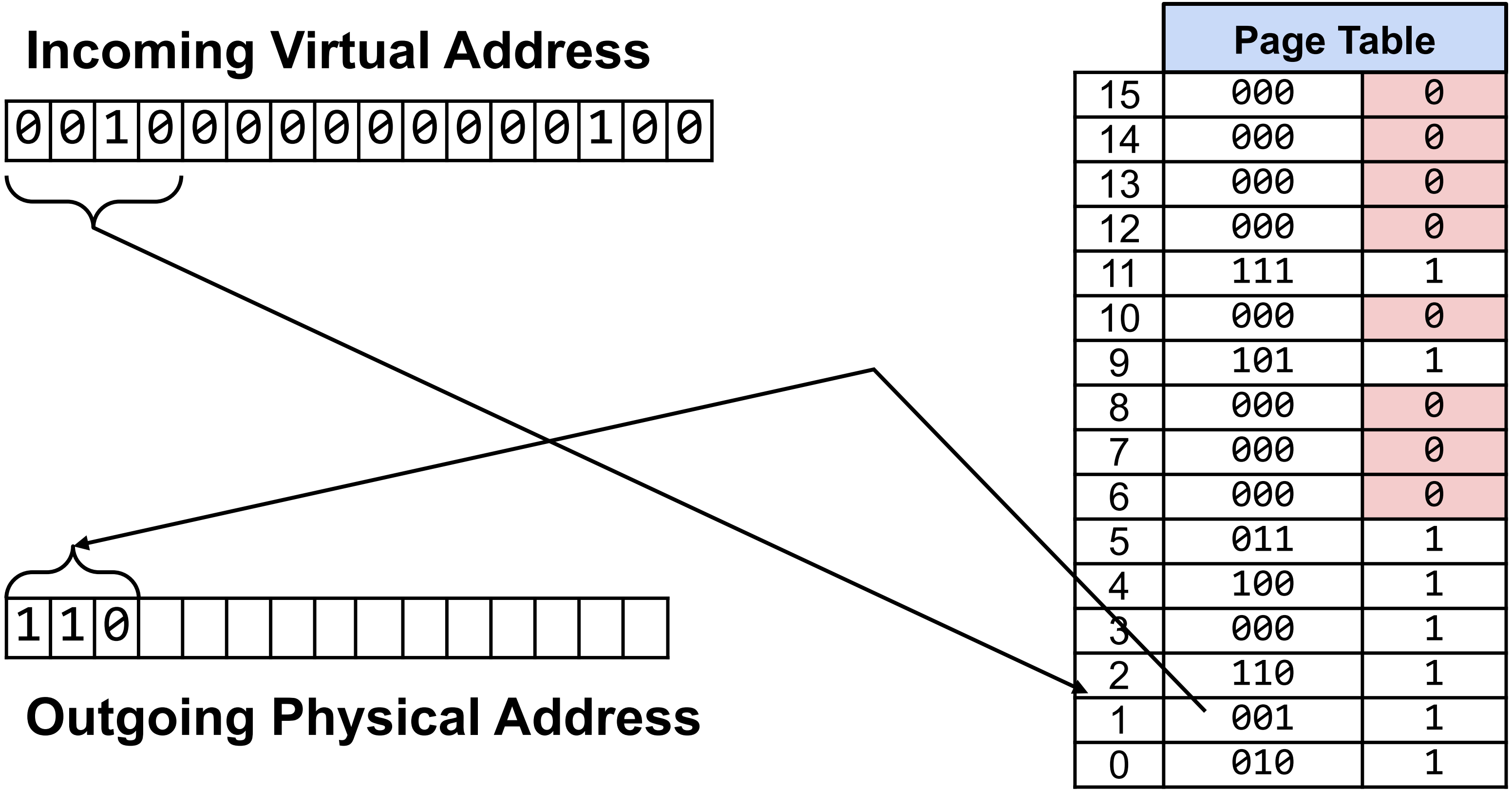
	Page Table	
15	000	0
14	000	0
13	000	0
12	000	0
11	111	1
10	000	0
9	101	1
8	000	0
7	000	0
6	000	0
5	011	1
4	100	1
3	000	1
2	110	1
1	001	1
0	010	1

Incoming Virtual Address

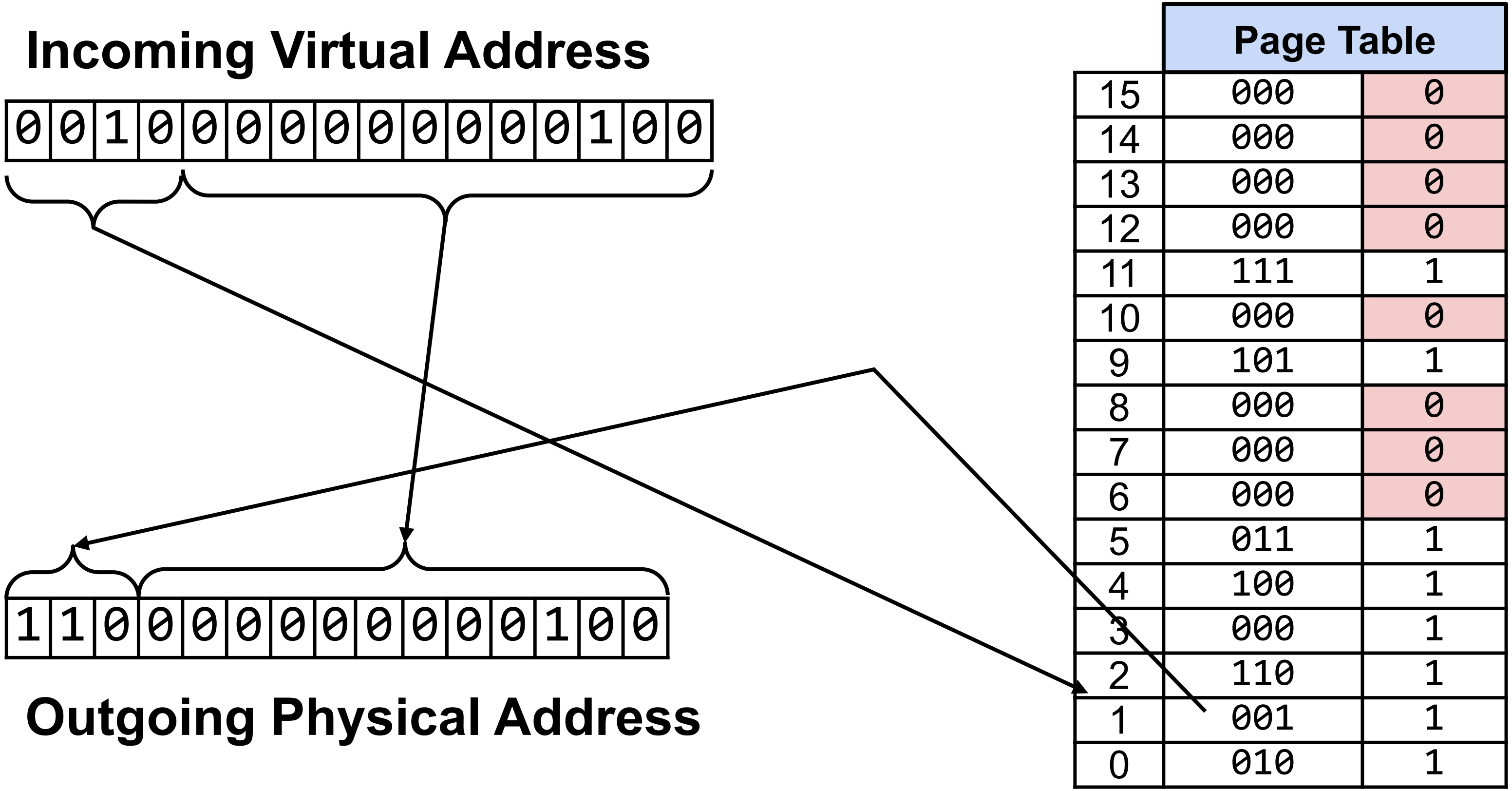
0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0

Outgoing Physical Address

	Page Table	
15	000	0
14	000	0
13	000	0
12	000	0
11	111	1
10	000	0
9	101	1
8	000	0
7	000	0
6	000	0
5	011	1
4	100	1
3	000	1
2	110	1
1	001	1
0	010	1



Incoming Virtual Address
0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0
Outgoing Physical Address
1 1 0
Page Table
15 000 0
14 000 0
13 000 0
12 000 0
11 111 1
10 000 0
9 101 1
8 000 0
7 000 0
6 000 0
5 011 1
4 100 1
3 000 1
2 110 1
1 001 1
0 010 1


Incoming Virtual Address
0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0
Outgoing Physical Address
1 1 0 0 0 0 0 0 0 0 0 0 1 0 0
Page Table
15 000 0
14 000 0
13 000 0
12 000 0
11 111 1
10 000 0
9 101 1
8 000 0
7 000 0
6 000 0
5 011 1
4 100 1
3 000 1
2 110 1
1 001 1
0 010 1

Incoming Virtual Address

1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0

What about this scenario?
Trace the steps

Outgoing Physical Address

	Page Table	
15	000	0
14	000	0
13	000	0
12	000	0
11	111	1
10	000	0
9	101	1
8	000	0
7	000	0
6	000	0
5	011	1
4	100	1
3	000	1
2	110	1
1	001	1
0	010	1

Incoming Virtual Address

1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0

	Page Table	
15	000	0
14	000	0
13	000	0
12	000	0
11	111	1
10	000	0
9	101	1
8	000	0
7	000	0
6	000	0
5	011	1
4	100	1
3	000	1
2	110	1
1	001	1
0	010	1

Outgoing Physical Address

Incoming Virtual Address

1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0

	Page Table	
15	000	0
14	000	0
13	000	0
12	000	0
11	111	1
10	000	0
9	101	1
8	000	0
7	000	0
6	000	0
5	011	1
4	100	1
3	000	1
2	110	1
1	001	1
0	010	1

Absent!
Must be swapped in

Outgoing Physical Address

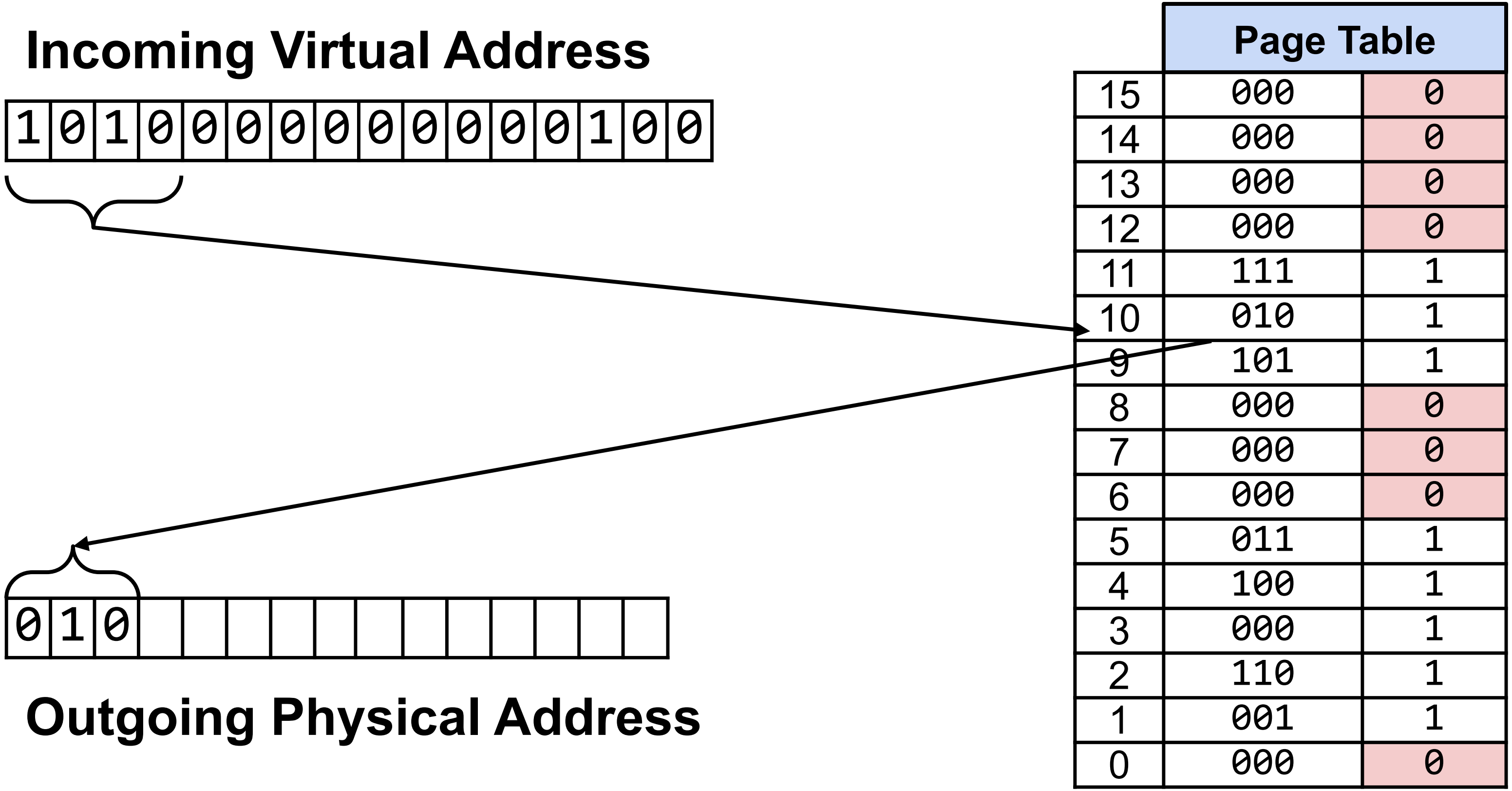
Incoming Virtual Address

1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0

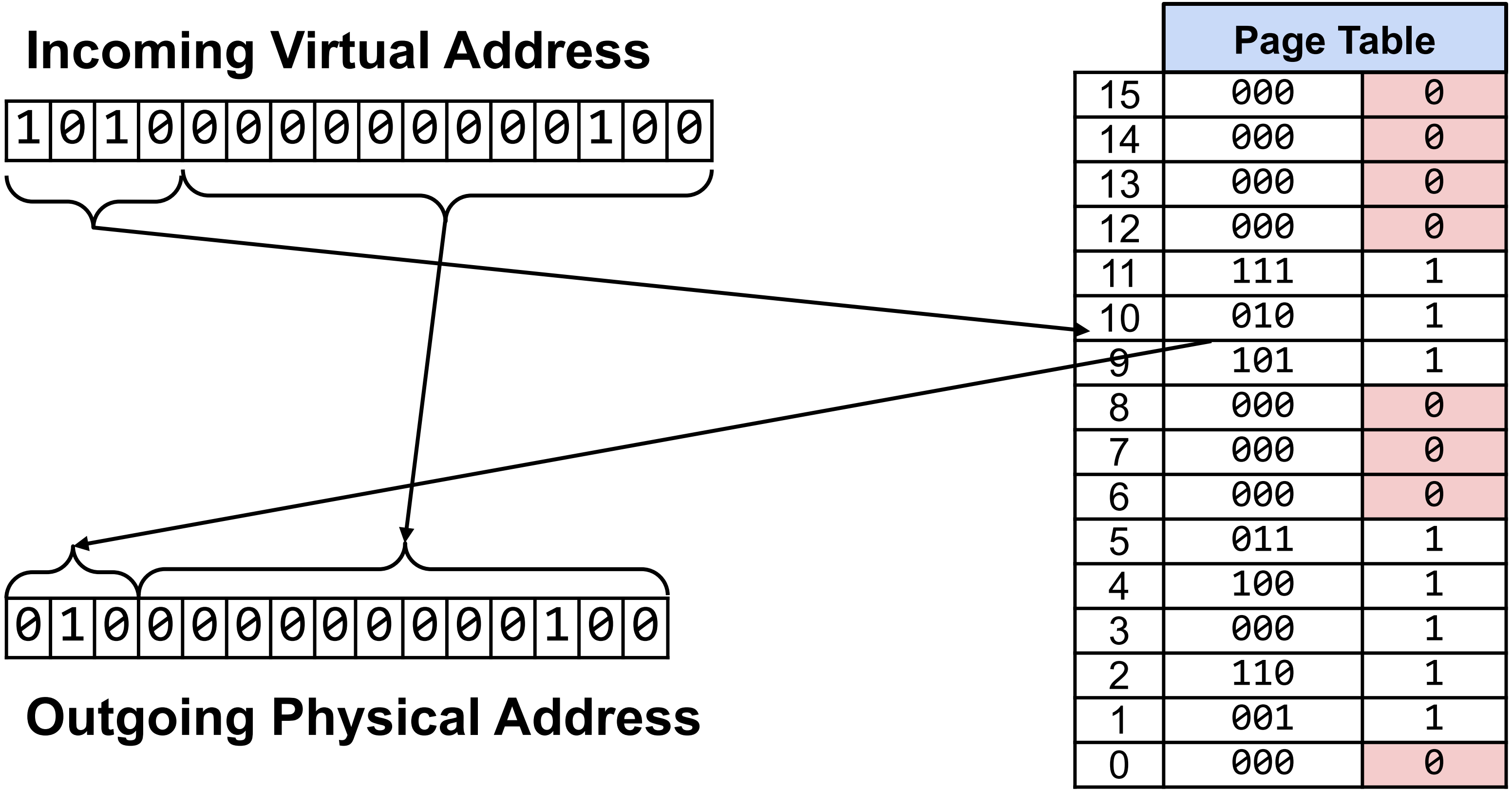
Absent!
Must be swapped in

Outgoing Physical Address

	Page Table	
15	000	0
14	000	0
13	000	0
12	000	0
11	111	1
10	010	1
9	101	1
8	000	0
7	000	0
6	000	0
5	011	1
4	100	1
3	000	1
2	110	1
1	001	1
0	000	0



Incoming Virtual Address
1 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0
Page Table
15 000 0
14 000 0
13 000 0
12 000 0
11 111 1
10 010 1
9 101 1
8 000 0
7 000 0
6 000 0
5 011 1
4 100 1
3 000 1
2 110 1
1 001 1
0 000 0
0 1 0
Outgoing Physical Address


Incoming Virtual Address
1 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0
Page Table
15 000 0
14 000 0
13 000 0
12 000 0
11 111 1
10 010 1
9 101 1
8 000 0
7 000 0
6 000 0
5 011 1
4 100 1
3 000 1
2 110 1
1 001 1
0 000 0
0 1 0 0 0 0 0 0 0 0 0 0 1 0 0
Outgoing Physical Address

Page Table Entry

- Each entry needs some information, in addition to present bit and the frame number
  - Protection: Permission to Read? Write? Exec?
  - Modification: Has this data been modified?
  - Referenced: Has this page been referenced?
  - Can this be cached (for optimization reasons)?

extra	caching en / dis	refd	mod	protect	present / absent	Frame Number

Page Tables - Two Implementations

- Have an in-hardware page table in the MMU (on or near the CPU). Whenever a context switch happens, load the processes page table into hardware, have MMU do all instruction translations
- Have a register in the CPU that points to the current processes page table. Update this register whenever there is a context switch happens, use MMU+Memory to translate addresses

Page Tables - Two Implementations

- Have an in-hardware page table in the MMU (on or near the CPU). Whenever a context switch happens, load the processes page table into hardware, have MMU do all instruction translations
- Have a register in the CPU that points to the current processes page table. Update this register whenever there is a context switch happens, use MMU+Memory to translate addresses

What are the pros and cons of each?

Page Tables - Two Implementations

- Dedicated PT Hardware
  - Pro: Once loaded, fast!
  - Cons: Context switching expensive, could be huge table
- Memory PT
  - Pro: Super cheap to context switch
  - Con: Slower overall execution

The cons in both cases are NOT acceptable for performance!

Starting a Process

- When a process first begins, how does the OS know where to get the memory from?
  - instructions, space for global vars, etc
- From the ELF file!
- Includes VMA (Virtual Memory Address) Information

ELF Sections, VMAs

bddicken — ssh bddicken@lectura.cs.arizona.edu — 170×47

```

a.out:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .interp       0000001c  0000000000000318  0000000000000318  00000318  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.gnu.property 00000020  0000000000000338  0000000000000338  00000338  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .note.gnu.build-id 00000024  0000000000000358  0000000000000358  00000358  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .note.ABI-tag 00000020  000000000000037c  000000000000037c  0000037c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .gnu.hash     00000024  00000000000003a0  00000000000003a0  000003a0  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .dynsym       00000090  00000000000003c8  00000000000003c8  000003c8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .dynstr       0000007d  0000000000000458  0000000000000458  00000458  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .gnu.version  0000000c  00000000000004d6  00000000000004d6  000004d6  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .gnu.version_r 00000020  00000000000004e8  00000000000004e8  000004e8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .rela.dyn     000000c0  0000000000000508  0000000000000508  00000508  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 10 .init         0000001b  0000000000001000  0000000000001000  00001000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 11 .plt          00000010  0000000000001020  0000000000001020  00001020  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
```

Fork and Exec

- What to do when we want to create a new process?
- In a UNIX System, call **`fork()`**
  - Create new entry in process table, get new PID, etc
  - What do do with all the memory?

Fork and Exec

- What to do when we want to create a new process?
- In a UNIX System, call `fork()`
  - Create new entry in process table, get new PID, etc
  - What do do with all the memory?
- Gets an exact memory copy!
- Does not need to copy it all at once, use Copy-On-Write instead.

Fork and Exec

- Often, After a fork occurs, there is a follow-up call to **exec**
- Exec replaces the memory for the current process
- Used in tandem to create new processes with new memories

Page Tables - Two Implementations

What hardware could be added to a memory-based page table system to provide a speed boost?

Use Dedicated Hardware + Memory

- Key observation: Most programs make a large number of references to a small number of pages, not the other way around
- Storing the entire page table in hardware is likely super excessive
- Instead, store a subset of page table entries in hardware
- **Translation Lookaside Buffers**!

Translation Lookaside Buffers

A **Translation Lookaside Buffer** (TLB) is a relatively small array-like hardware structure that stores a small number of page table entries. Ideally, ones that are actively / often being used by the CPU.

For example, TLB could store 32 PT entries. If former observation is true, the majority of memory references will use this instead of needing to look at in-memory PT.

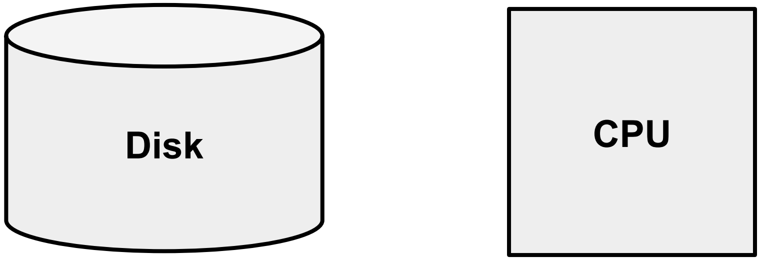
Program A

```
print('hi')
let arr;

def calc(x):
  i = 0:
  while i < 10000:
    arr[i%100] = x
    i++

def main():
  arr = alloc(100)
  calc(5)
  calc(10)

main()
```



Page Table

60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

TLB

Valid	Virt Page	Mod	Prot	Page Frame

Physical Memory

28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

```
print('hi')
let arr;

def calc(x):
  i = 0:
  while i < 10000:
    arr[i%100] = x
    i++

def main():
  arr = alloc(100)
  calc(5)
  calc(10)

main()
```

Time to load from disk

Disk

CPU

Not here!

Page Table	
60-64k	X
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Not here!

TLB

Valid	Virt Page	Mod	Prot	Page Frame

Physical Memory

28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

```
print('hi')
let arr;

def calc(x):
  i = 0:
  while i < 10000:
    arr[i%100] = x
    i++

def main():
  arr = alloc(100)
  calc(5)
  calc(10)

main()
```

Load

Disk

CPU

Choose a slot

Page Table	
60-64k	0
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Update TLB

TLB

Valid	Virt Page	Mod	Prot	Page Frame
1	15	0	RX	0

Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

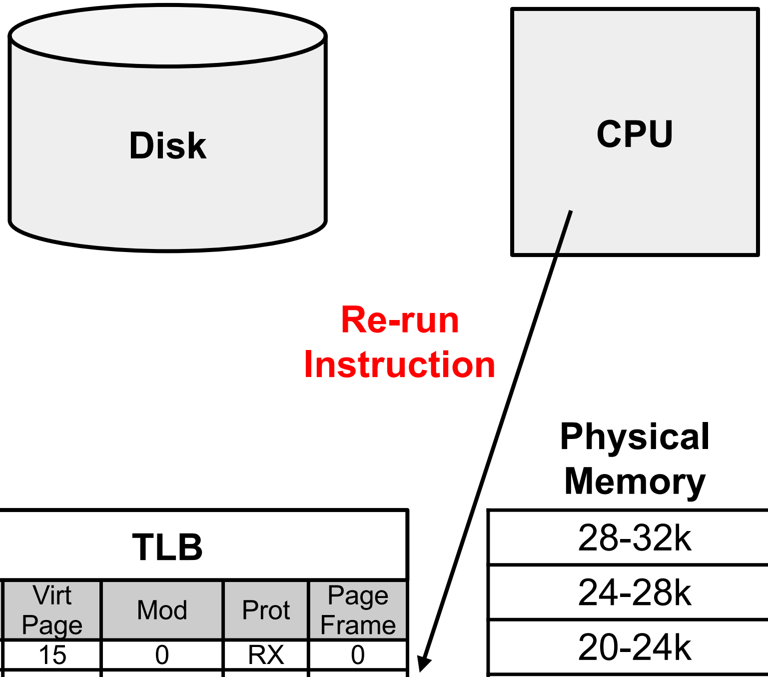
```
print('hi')
let arr;

def calc(x):
  i = 0:
  while i < 10000:
    arr[i%100] = x
    i++

def main():
  arr = alloc(100)
  calc(5)
  calc(10)

main()
```

Page Table	
60-64k	0
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X



TLB

Valid	Virt Page	Mod	Prot	Page Frame
1	15	0	RX	0

Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

```
print('hi')
let arr;

def calc(x):
  i = 0:
  while i < 10000:
    arr[i%100] = x
    i++

def main():
  arr = alloc(100)
  calc(5)
  calc(10)

main()
```

This probably all fit in that one page. Now, all instructions loaded into memory.

Page Table	
60-64k	0
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Disk

CPU

TLB

Valid	Virt Page	Mod	Prot	Page Frame
1	15	0	RX	0

Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

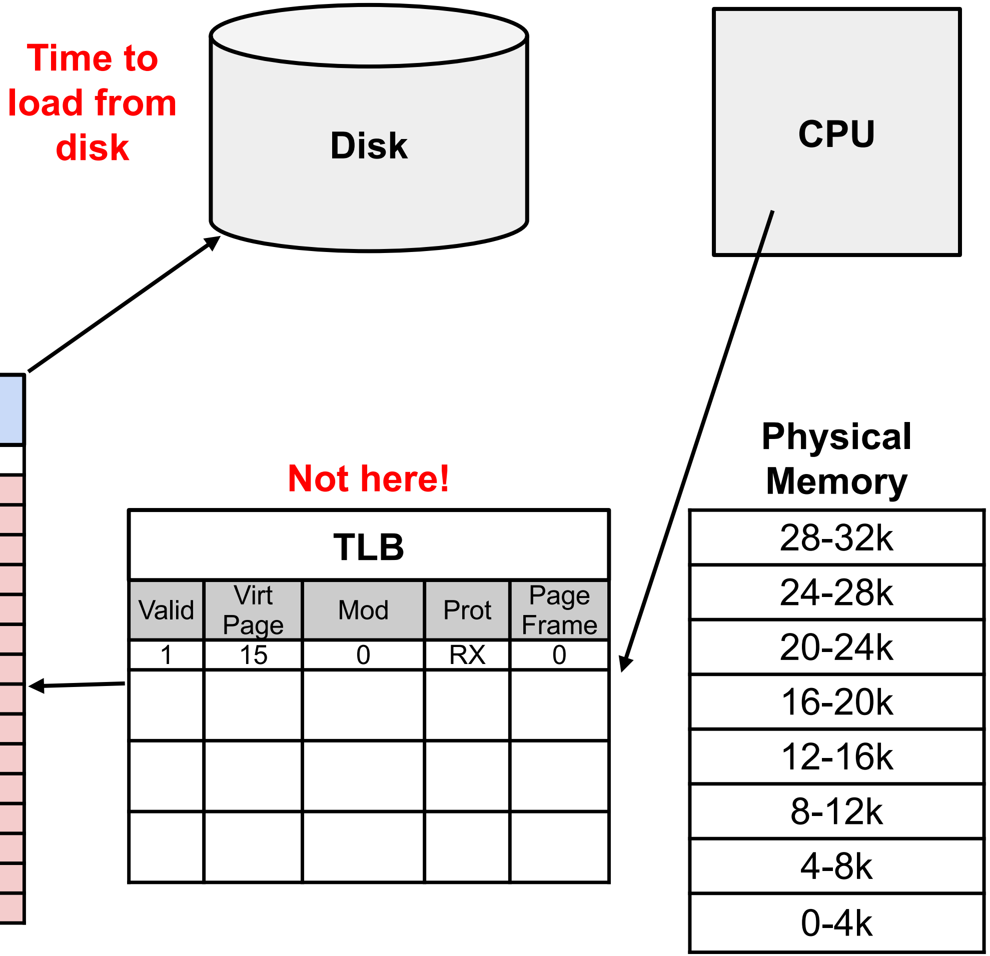
Program A

```
print('hi')
let arr;

def calc(x):
  i = 0:
  while i < 10000:
    arr[i%100] = x
    i++

def main():
  arr = alloc(100)
  calc(5)
  calc(10)

main()
```



Page Table	
60-64k	0
56-60k	X
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

TLB

Valid	Virt Page	Mod	Prot	Page Frame
1	15	0	RX	0

Physical Memory

28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

```
print('hi')
let arr;

def calc(x):
  i = 0:
  while i < 10000:
    arr[i%100] = x
    i++

def main():
  arr = alloc(100)
  calc(5)
  calc(10)

main()
```

Load

Disk

CPU

Choose a slot

Page Table	
60-64k	0
56-60k	1
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Update TLB

TLB

Valid	Virt Page	Mod	Prot	Page Frame
1	15	0	RX	0
1	14	0	RW	1

Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

```
print('hi')
let arr;

def calc(x):
  i = 0:
  while i < 10000:
    arr[i%100] = x
    i++

def main():
  arr = alloc(100)
  calc(5)
  calc(10)

main()
```

Is should be enough space for the stack as this program executes!

Disk

CPU

Page Table	
60-64k	0
56-60k	1
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	X
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

TLB

Valid	Virt Page	Mod	Prot	Page Frame
1	15	0	RX	0
1	14	0	RW	1

Physical Memory

28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

```
print('hi')
let arr;

def calc(x):
  i = 0:
  while i < 10000:
    arr[i%100] = x
    i++

def main():
  arr = alloc(100)
  calc(5)
  calc(10)

main()
```

Page Table	
60-64k	0
56-60k	1
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	2
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Disk

CPU

Update TLB

TLB

Valid	Virt Page	Mod	Prot	Page Frame
1	15	0	RX	0
1	14	0	RW	1

Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

```
print('hi')
let arr;

def calc(x):
  i = 0:
  while i < 10000:
    arr[i%100] = x
    i++

def main():
  arr = alloc(100)
  calc(5)
  calc(10)

main()
```

Disk

CPU

Choose a slot

Page Table	
60-64k	0
56-60k	1
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	4
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Update TLB

TLB

Valid	Virt Page	Mod	Prot	Page Frame
1	15	0	RX	0
1	14	0	RW	1
1	7	0	RW	4

Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

```
print('hi')
let arr;

def calc(x):
  i = 0:
  while i < 10000:
    arr[i%100] = x
    i++

def main():
  arr = alloc(100)
  calc(5)
  calc(10)

main()
```

This should fit entire array!

Disk

CPU

Choose a slot

Page Table	
60-64k	0
56-60k	1
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	4
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

Update TLB

TLB

Valid	Virt Page	Mod	Prot	Page Frame
1	15	0	RX	0
1	14	0	RW	1
1	7	0	RW	4

Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

```
print('hi')
let arr;

def calc(x):
  i = 0:
  while i < 10000:
    arr[i%100] = x
    i++

def main():
  arr = alloc(100)
  calc(5)
  calc(10)

main()
```

From here on out, should be mostly TLB hits as CPU executes instructions.

CPU

Page Table	
60-64k	0
56-60k	1
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	4
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

TLB

Valid	Virt Page	Mod	Prot	Page Frame
1	15	0	RX	0
1	14	0	RW	1
1	7	0	RW	4

Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Program A

```
print('hi')
let arr;

def calc(x):
  i = 0:
  while i < 10000:
    arr[i%100] = x
    i++

def main():
  arr = alloc(100)
  calc(5)
  calc(10)

main()
```

What about Context Switches?
What happens to the TLB?

CPU

Page Table	
60-64k	0
56-60k	1
52-56k	X
48-52k	X
44-48k	X
40-44k	X
36-40k	X
32-36k	X
28-32k	4
24-28k	X
20-24k	X
16-20k	X
12-16k	X
8-12k	X
4-8k	X
0-4k	X

TLB

Valid	Virt Page	Mod	Prot	Page Frame
1	15	0	RX	0
1	14	0	RW	1
1	7	0	RW	4

Physical Memory
28-32k
24-28k
20-24k
16-20k
12-16k
8-12k
4-8k
0-4k

Hardware vs Software Handling

- Could have the MMU hardware handle all TLB misses directly
- Or, could have an interrupt occur whenever TLB miss happens, have kernel handle the issue

Hardware vs Software Handling

- Could have the MMU hardware handle all TLB misses directly
- Or, could have an interrupt occur whenever TLB miss happens, have kernel handle the issue

What are the pros and cons of each?

TLB Hits and Misses

- **Hit**: A virtual memory address is in TLB
- **Soft Miss**: A virtual memory address is not in TLB, but is in memory
- **Hard Miss**: A virtual memory address lookup is not in TLB or in memory, go to disk instead

TLB Hits and Misses

- **Hit**: A virtual memory address is in TLB
- **Soft Miss**: A virtual memory address is not in TLB, but is in memory
- **Hard Miss**: A virtual memory address lookup is not in TLB or in memory, go to disk instead

Performance of each?

Page Replacement Algorithms

When it comes to to evict a page out of physical memory and replace with another, how to make this choice?

What algorithms can you think of that might work well?

MMAP

```c
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
int main() {
  int fd = open("/bin/bash", O_RDONLY);
  printf("fd %d\n", fd);
  char *buf = mmap(NULL, 4096,
                   PROT_READ,
                   MAP_PRIVATE,
                   fd, 0);
 printf("The first 8 bytes are:\n");
 for (int i=0; i<8; i++)
 printf(" %d: 0x%02x\n", i, buf[i]);
 return 0;
}
```

MMAP

```c
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
int main() {
  int fd = open("/bin/bash", O_RDONLY);
  printf("fd %d\n", fd);
  char *buf = mmap(NULL, 4096,
                   PROT_READ,
                   MAP_PRIVATE,
                   fd, 0);
 printf("The first 8 bytes are:\n");
 for (int i=0; i<8; i++)
 printf(" %d: 0x%02x\n", i, buf[i]);
 return 0;
}
```

open returns a file descriptor, needed fo using mmap

MMAP

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
int main() {
  int fd = open("/bin/bash", O_RDONLY);
  printf("fd %d\n", fd);
  char *buf = mmap(NULL, 4096,
                   PROT_READ,
                   MAP_PRIVATE,
                   fd, 0);
 printf("The first 8 bytes are:\n");
 for (int i=0; i<8; i++)
 printf(" %d: 0x%02x\n", i, buf[i]);
 return 0;
}
```

NULL for address, one page (4094)

Create a read-only page

MMAP

```c
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
int main() {
  int fd = open("/bin/bash", O_RDONLY);
  printf("fd %d\n", fd);
  char *buf = mmap(NULL, 4096,
                 PROT_READ,
                 MAP_PRIVATE,
                 fd, 0);
 printf("The first 8 bytes are:\n");
 for (int i=0; i<8; i++)
 printf(" %d: 0x%02x\n", i, buf[i]);
 return 0;
}
```

This copy is only for this process to use, not shared

Use the fd file, start at byte 0

MMAP

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
int main() {
  int fd = open("/bin/bash", O_RDONLY);
  printf("fd %d\n", fd);
  char *buf = mmap(NULL, 4096,
                   PROT_READ | PROT_WRITE,
                   MAP_SHARED,
                   fd, 0);
 printf("The first 8 bytes are:\n");
 for (int i=0; i<8; i++)
 printf(" %d: 0x%02x\n", i, buf[i]);
 return 0;
}
```

Can also configure to be shareable!

Other processes forked from this one will be able to see the same page

MMap and MProtect Example

1. Copy **`/tmp/m_sharing_protecting.c`** to a directory you own
2. Read the code. Can you tell me what it does?
3. Create a file named **`testing.txt`** in the same directory, compile, run
4. What will be in **`testing.txt`** after it runs?

MMap and MProtect Example

What happens is the parent processes tries to write something to the shared buffer?

Virtual Memory

1. Copy **`/tmp/m_small.c`** to a directory you own
2. Read the code. Can you tell me what it does?

Virtual Memory

1. Copy **`/tmp/m_small.c`** to a directory you own
2. Read the code. Can you tell me what it does?

3. Now, do the same for **`/tmp/m_large.c`**. What is the difference in the code? Performance difference?

Virtual Memory

1. Copy **`/tmp/m_small.c`** to a directory you own
2. Read the code. Can you tell me what it does?

3. Now, do the same for **`/tmp/m_large.c`**. What is the difference in the code? Performance difference?

4. Now, do the same for **`/tmp/m_multiple_passes.c`**.

Checking for Page Table Misses

- How to look at the page table misses?
    - min ≈ soft miss and maj ≈ hard miss

```
$ ps -eo min_flt,maj_flt,cmd | grep -e m_reuse -e MINFL
```

Checking for Page Table Misses

- How to look at the page table misses?
  - min ≈ soft miss and maj ≈ hard miss

```
$ ps -eo min_flt,maj_flt,cmd | grep -e m_reuse -e MINFL
```

- The min_flt and maj_flt can be viewed for whatever running processes you want.

Attribution

Several examples in these slides were inspired by ones from "Modern Operating Systems" by Andrew Tanenbaum