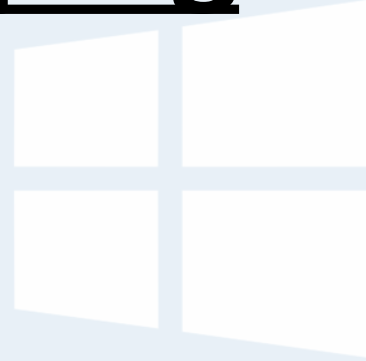


Operating Systems

COMS(3010A)

Swapping

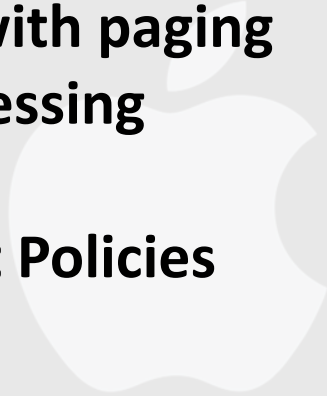
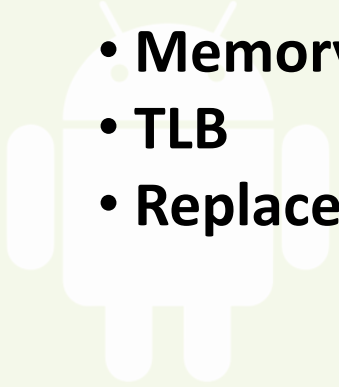


Branden Ingram

branden.ingram@wits.ac.za

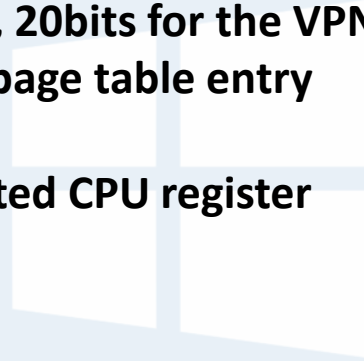
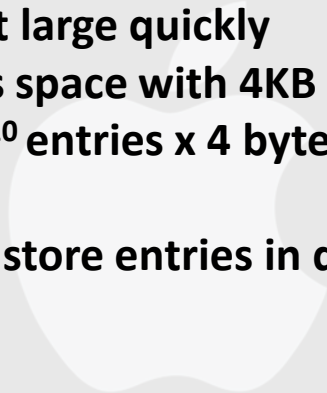
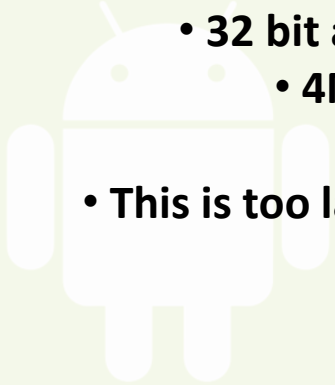
Recap

- **Paging**
- **Translation with paging**
- **Memory accessing**
- **TLB**
- **Replacement Policies**



Ok well where are these tables stored?

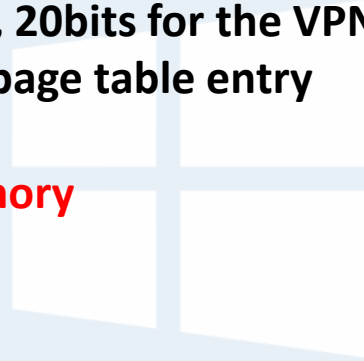
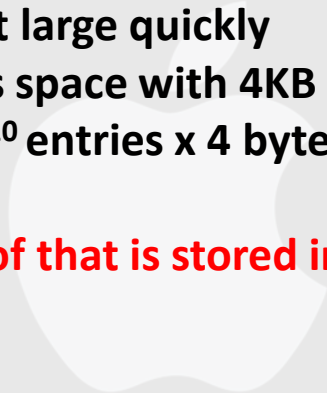
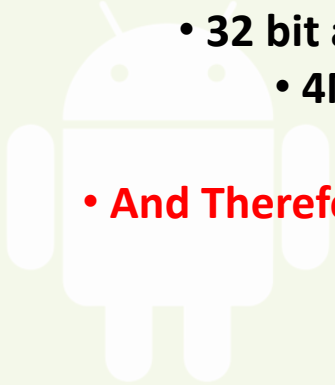
- Page tables can get large quickly
 - 32 bit address space with 4KB pages, 20bits for the VPN
 - $4\text{MB} = 2^{20}$ entries x 4 bytes per page table entry
- This is too large to store entries in dedicated CPU register



Ok well where are these tables stored?

- Page tables can get large quickly
 - 32 bit address space with 4KB pages, 20bits for the VPN
 - $4\text{MB} = 2^{20}$ entries x 4 bytes per page table entry

• **And Therefore all of that is stored in memory**



Ok well where are these tables stored?

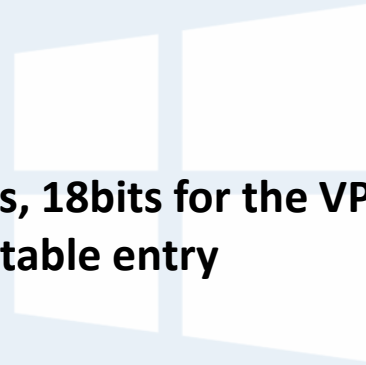
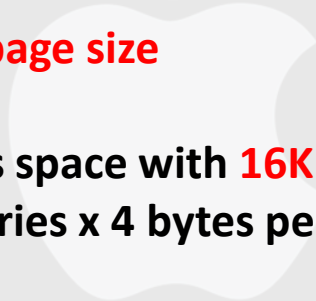
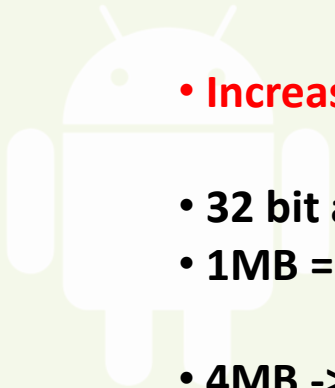
- Page tables can get large quickly
 - 32 bit address space with 4KB pages, 20bits for the VPN
 - $4\text{MB} = 2^{20}$ entries x 4 bytes per page table entry
- And Therefore all of that is stored in memory
- Recall also: we usually have **one page table for every process** in the system
 - With a **100** active processes (not uncommon on a modern system),
 - we will be allocating hundreds of megabytes of memory just for page tables!

So how can we reduce this burden?

- 32 bit address space with 4KB pages, 20bits for the VPN
- 4MB = 2^{20} entries x 4 bytes per page table entry

- **Increase the page size**

- 32 bit address space with **16KB** pages, 18bits for the VPN
- 1MB = 2^{18} entries x 4 bytes per page table entry
- 4MB -> 1MB (Naive Solution)



A hybrid Approach

- Combine Segmentation and Paging

Segment Register

Segment	Base	Size
Code	256KB	4KB
Heap	260KB	60KB
Stack	128KB	64KB



Page Table

Page	Page Frame
0	3
1	7
2	5
3	2

A hybrid Approach

- Instead of having a single page table for the entire address space of the process, why not have one per logical segment?
- In this example, we might thus have three page tables, one for the code, heap, and stack parts of the address space.

Segment Register

Segment	Base	Size
Code	256KB	4KB
Heap	260KB	60KB
Stack	128KB	64KB

Page Table

Page Table	
Page	Page Frame
0	3
1	7
2	5
3	2

A hybrid Approach

- Now, remember with segmentation, we had a base register that told us where each segment lived in physical memory, and a bound or limit register that told us the size of said segment.
- In our hybrid, we still have those structures in the MMU; here, we use the base not to point to the segment itself but rather to hold the physical address of the page table of that segment. The bounds register is used to indicate the end of the page table (i.e., how many valid pages it has).



Example

- Assume a 14-bit virtual address space with 512-byte pages,
- Address space split into four segments.
- Physical space is 65536-bytes

Segment Register

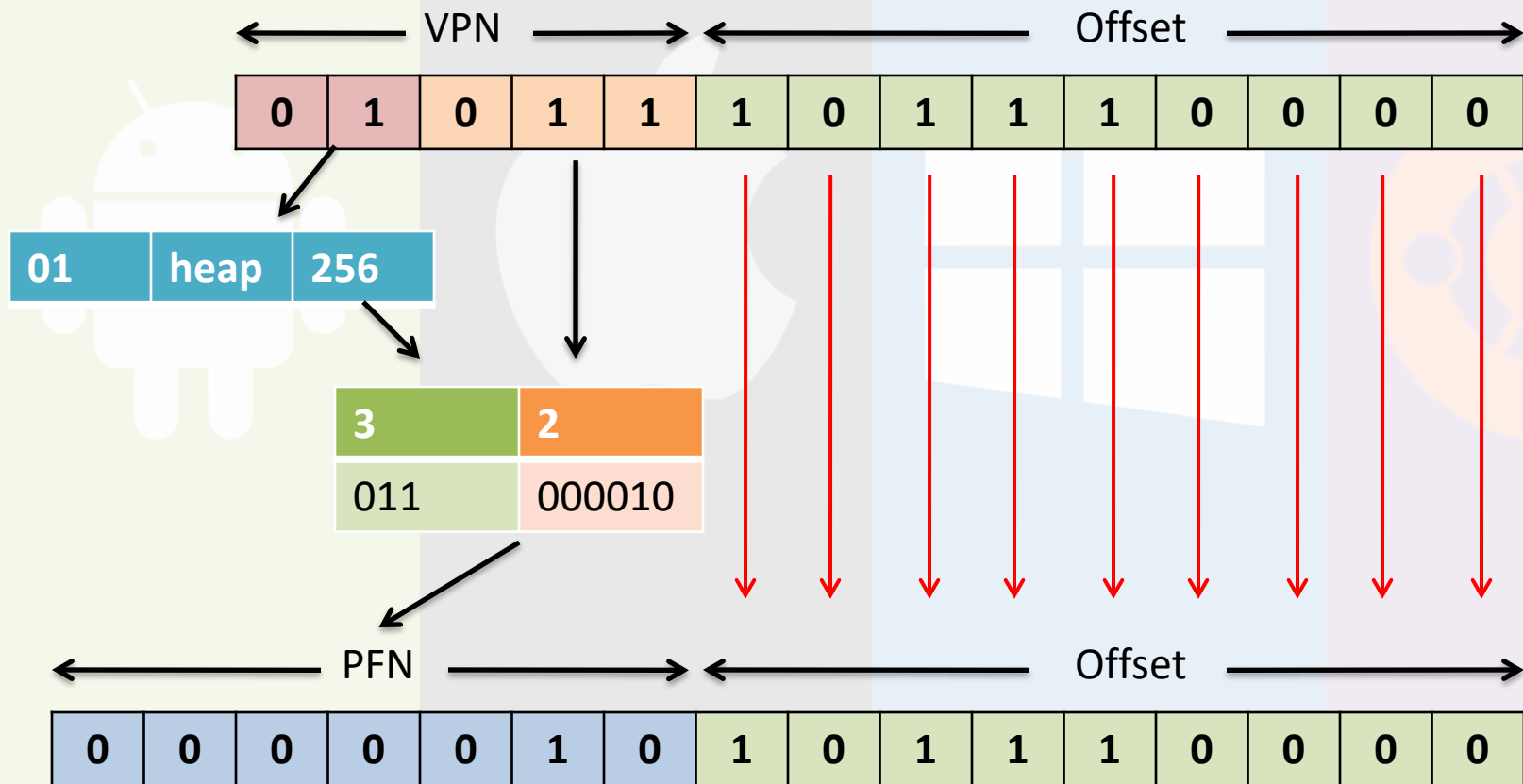
Segment	Base	Size
Code	12KB	4
Heap	3KB	4
Stack	7KB	4

Page Table

Page Table	
Page	Page Frame
0	3
1	7
2	5
3	2

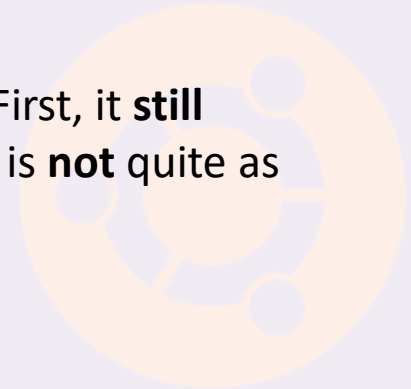
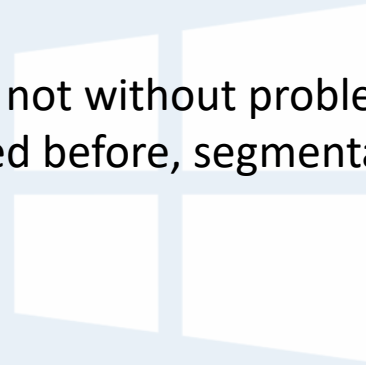
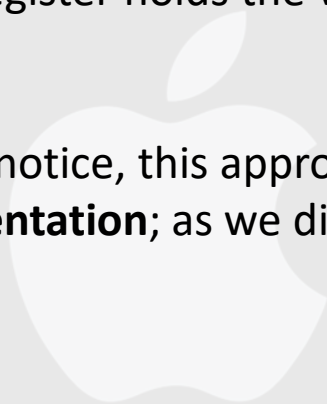
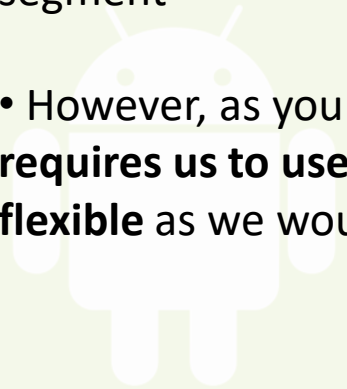
Address Translation : Example

- Example: virtual address 6000 in 16KB address space



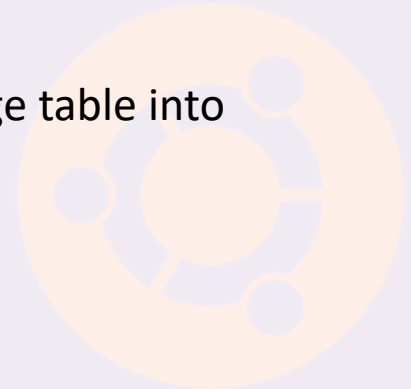
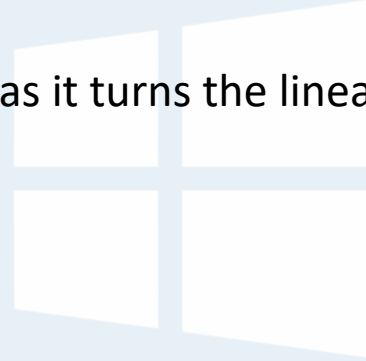
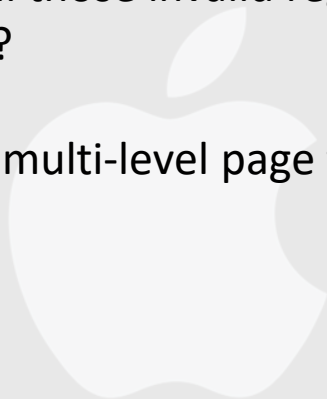
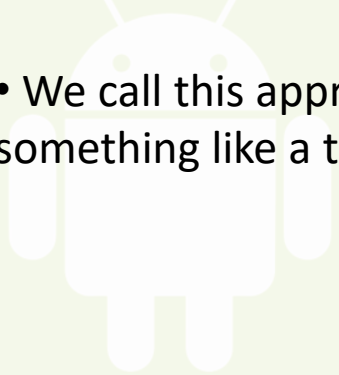
A hybrid Approach

- The critical difference in our hybrid scheme is the presence of a **bounds** register per segment; each bounds register holds the value of the **maximum valid page** in the segment
- However, as you might notice, this approach is not without problems. First, it **still requires us to use segmentation**; as we discussed before, segmentation is **not** quite as **flexible** as we would like



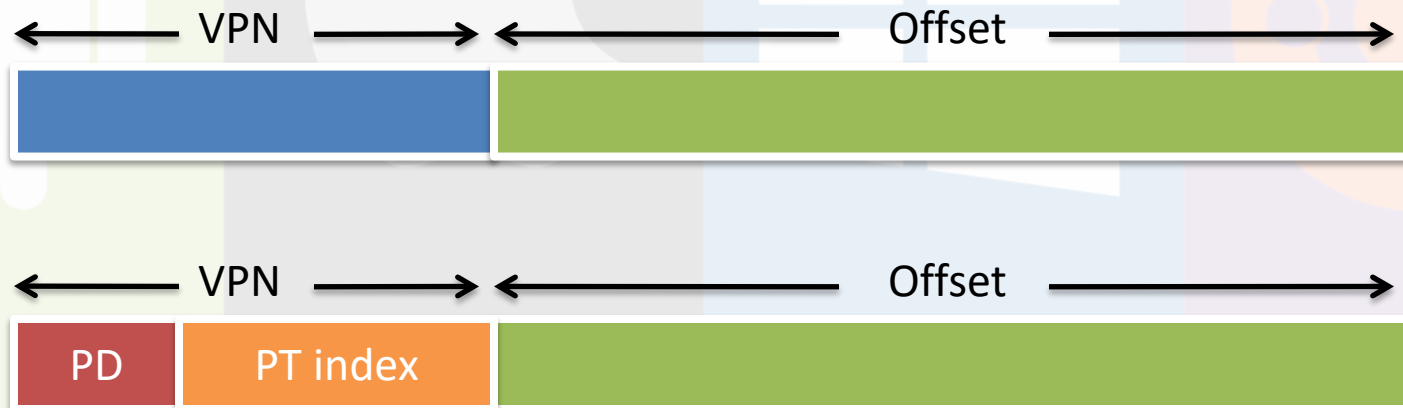
Multi-level Paging

- A different approach doesn't rely on segmentation but attacks the same problem:
 - how to get rid of all those invalid regions in the page table instead of keeping them all in memory?
- We call this approach a multi-level page table, as it turns the linear page table into something like a tree



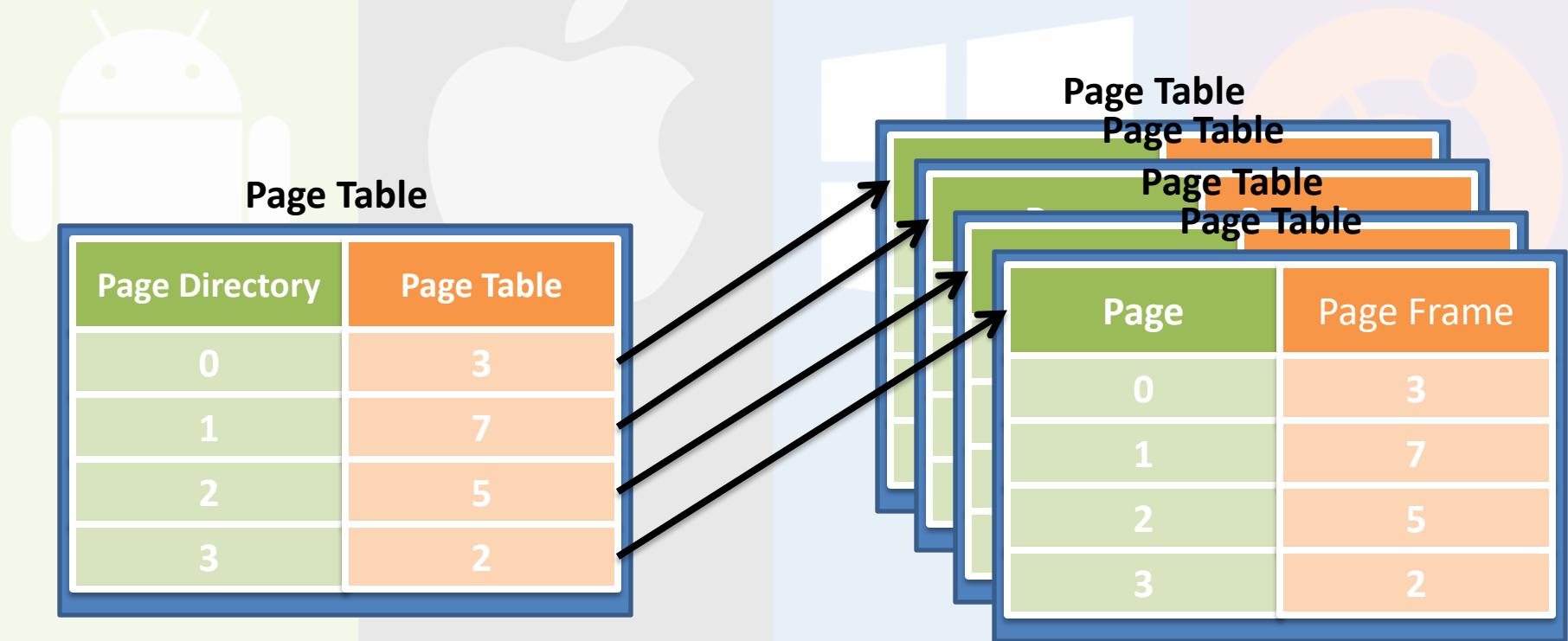
Multi-level Paging

- The basic idea behind a multi-level page table is simple. First, chop up the page table multiple levels of smaller tables, creating a hierarchical structure



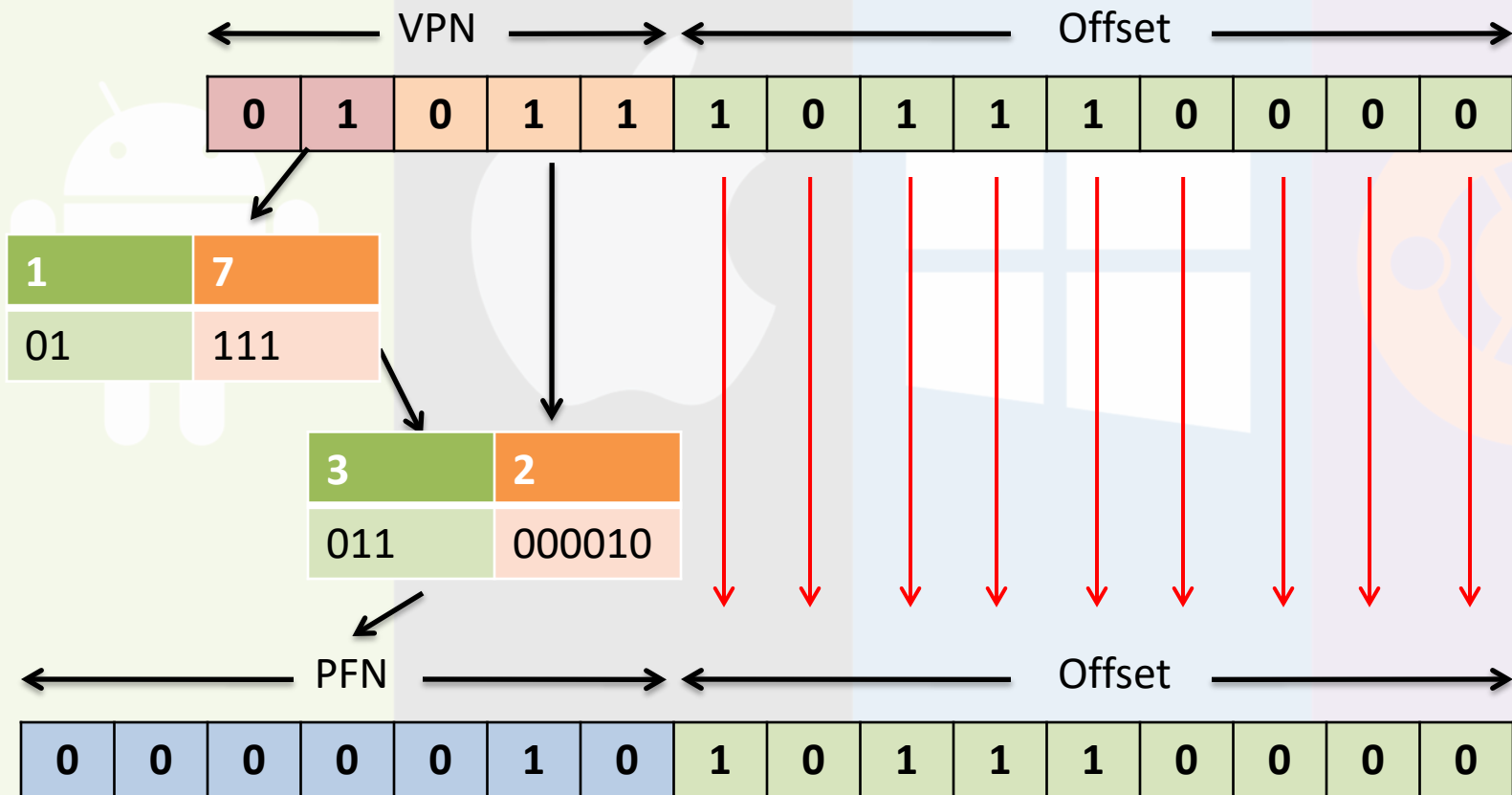
Multi-level Paging

- The basic idea behind a multi-level page table is simple. First, chop up the page table multiple levels of smaller tables, creating a hierarchical structure



Address Translation : Example

- Example: virtual address 6000 in 16KB address space



Comparison to Standard Paging

Standard Paging

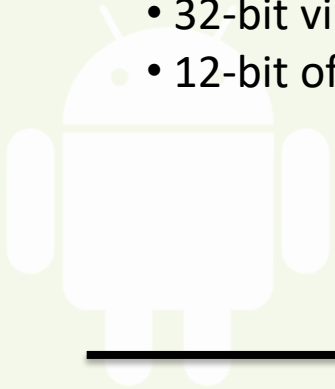
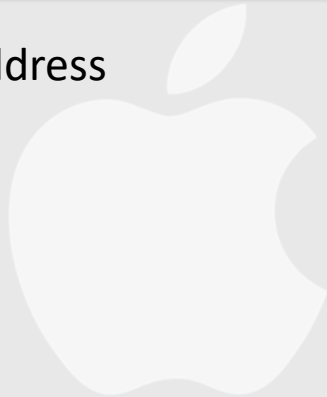
- 32-bit virtual address
- 12-bit offset

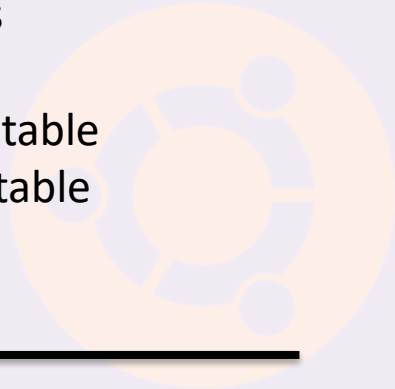
Multi-Level Paging

- 32-bit virtual address
- 12-bit offset
- 10-bit top level page table
- 10-bit 2nd level page table

Comparison to Standard Paging

Multi-Level Paging

- 
- 
- 32-bit virtual address
 - 12-bit offset

- 
- 32-bit virtual address
 - 12-bit offset
 - 10-bit top level page table
 - 10-bit 2nd level page table

How many possible Pages

- $32 - 12(\text{offset}) = 20\text{bits for VPN}$
- $2^{(20)} = 1,048,576$ pages

- The top-level page table has 2^{10} (1024) entries.
- Each entry points to a second-level page table, which also has 2^{10} entries.
- $2^{10} \times 2^{10} = 2^{20} = 1,048,576$ pages

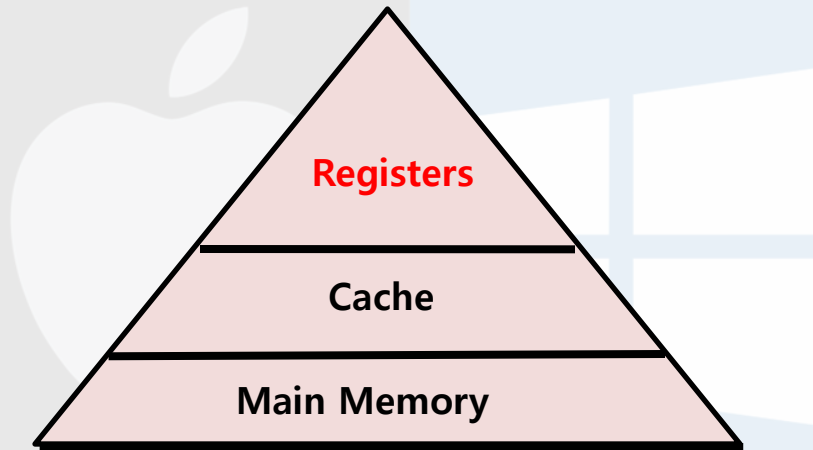
Advantages

- **Reduced Memory Overhead:** Instead of needing a large contiguous block of memory for a single-level page table, multi-level paging only requires memory for the tables that are actually used, which can be sparse.
- **Scalability:** It handles large address spaces more efficiently, especially in systems with large amounts of RAM or using 64-bit addressing.
- **Flexibility:** Multi-level paging can adapt to different sizes of memory by adding more levels or changing the sizes of the tables at each level.

Disadvantages

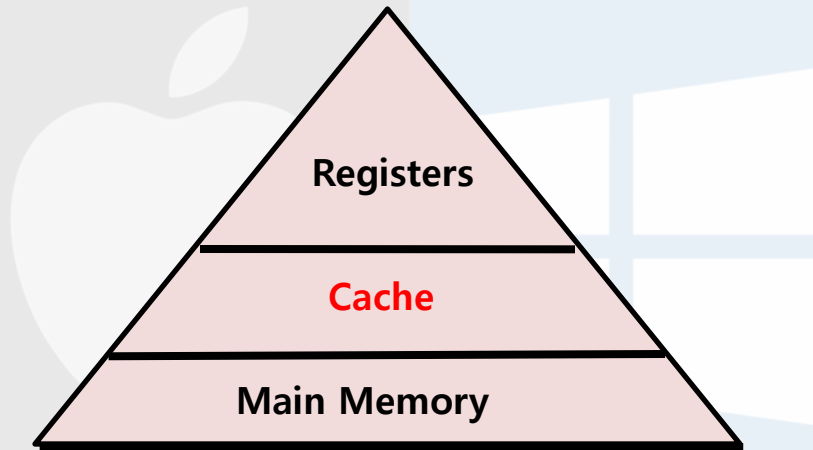
- **Increased Complexity:** The hierarchical structure adds complexity to memory management.
- **Additional Overhead:** Every memory access may involve multiple page table lookups, which can slow down address translation. However, modern processors often mitigate this with Translation Lookaside Buffers (TLBs) that cache recent address translations.

Swapping



Memory Hierarchy in modern system

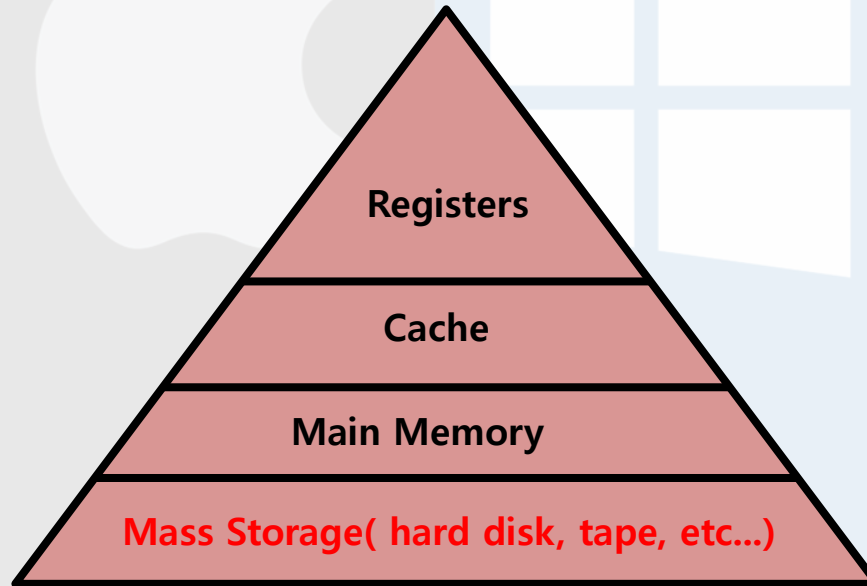
Swapping



Memory Hierarchy in modern system

Swapping

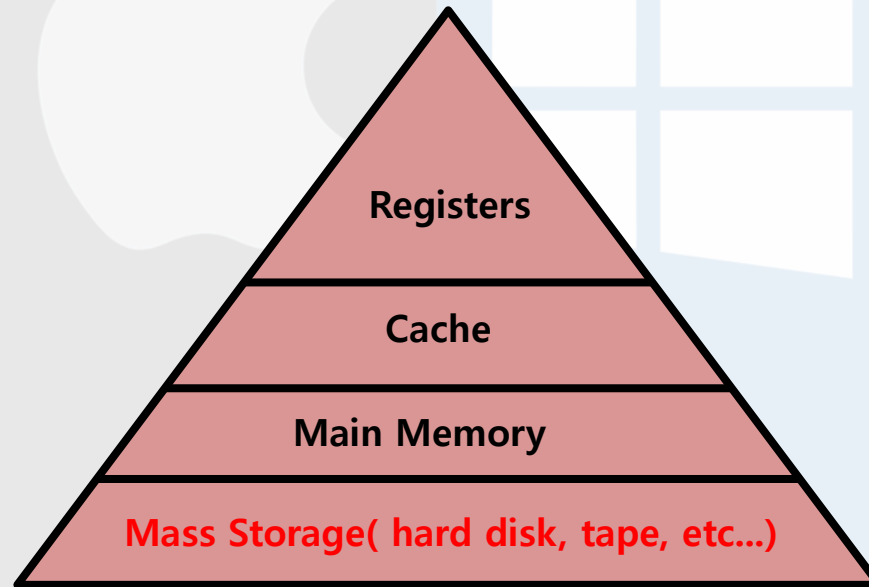
- Swapping involves us expanding our hierarchy out a bit to include long-term persistent storage.



Memory Hierarchy in modern system

Swapping

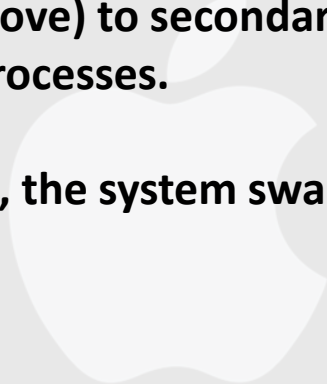
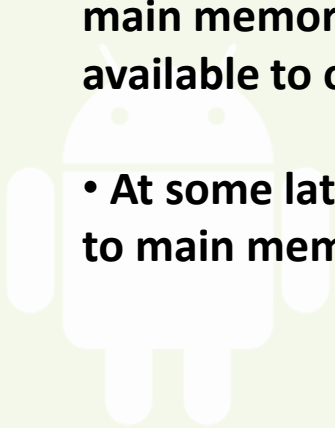
- OS need a place to stash away portions of address space that currently aren't in great demand.
- In modern systems, this role is usually served by a hard disk drive



Memory Hierarchy in modern system

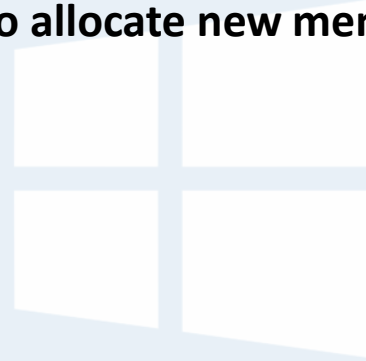
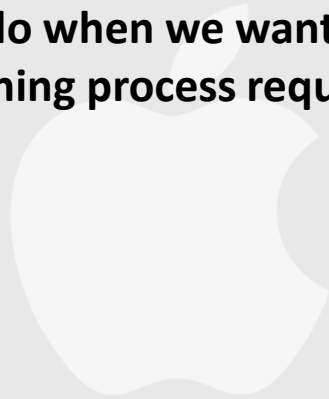
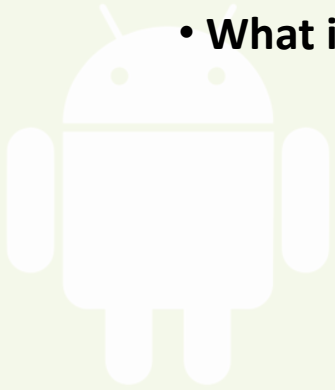
What is Swapping

- Swapping is a mechanism in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make that memory available to other processes.
- At some later time, the system swaps back the process from the secondary storage to main memory.



Why would we need swapping

- What if the memory is full
 - What do we do when we want to run a new program
 - What if a running process requests to allocate new memory



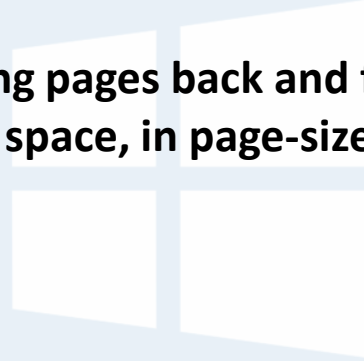
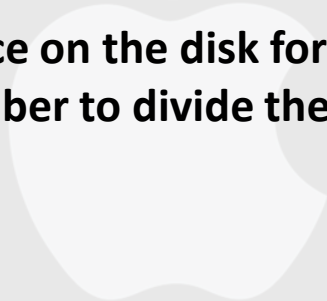
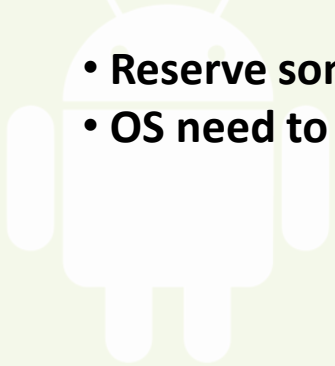
Swap Space

- Swapping makes use of a **swap space** which allows the OS to support the illusion of a large virtual memory for multiple concurrently-running processes



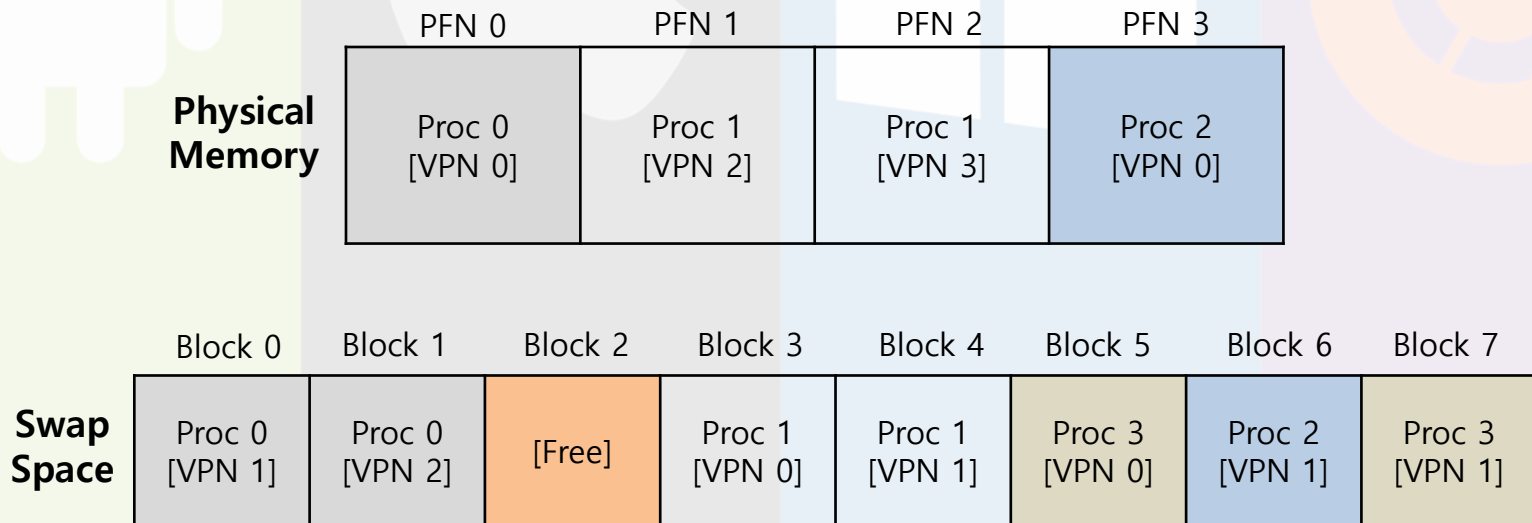
Swap Space

- Swapping makes use of a **swap space** which allows the OS to support the illusion of a large virtual memory for multiple concurrently-running processes
- Reserve some space on the disk for moving pages back and forth.
- OS need to remember to divide the swap space, in page-sized unit



Swap Space

- Swapping makes use of a **swap space** which allows the OS to support the illusion of a large virtual memory for multiple concurrently-running processes
- Reserve some space on the disk for moving pages back and forth.
- OS need to remember to divide the swap space, in page-sized unit



Physical Memory and Swap Space

How do we keep track of where these pages are then?

- Is a page in memory or on disk?



Present Bit

- Add some machinery higher up in the system in order to support swapping pages to and from the disk.

- When the hardware looks in the PTE, it may find that the page is not present in physical memory.

Value	Meaning
1	page is present in physical memory
0	The page is not in memory but rather on disk.

What if we try to access a page not in memory?

- The Page Fault

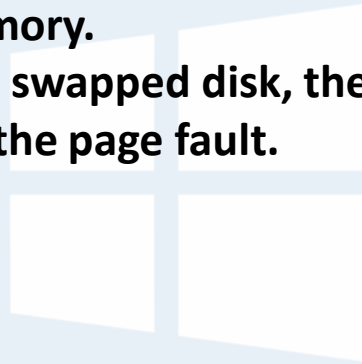
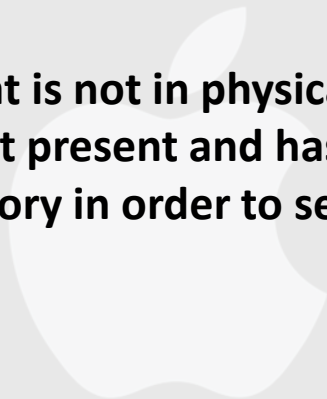
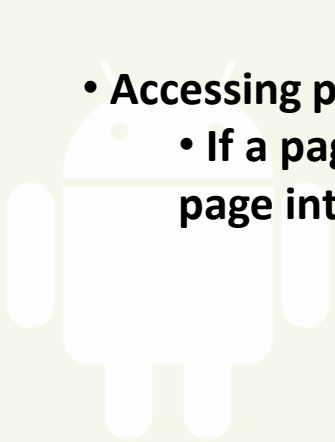


What if we try to access a page not in memory?

- The Page Fault

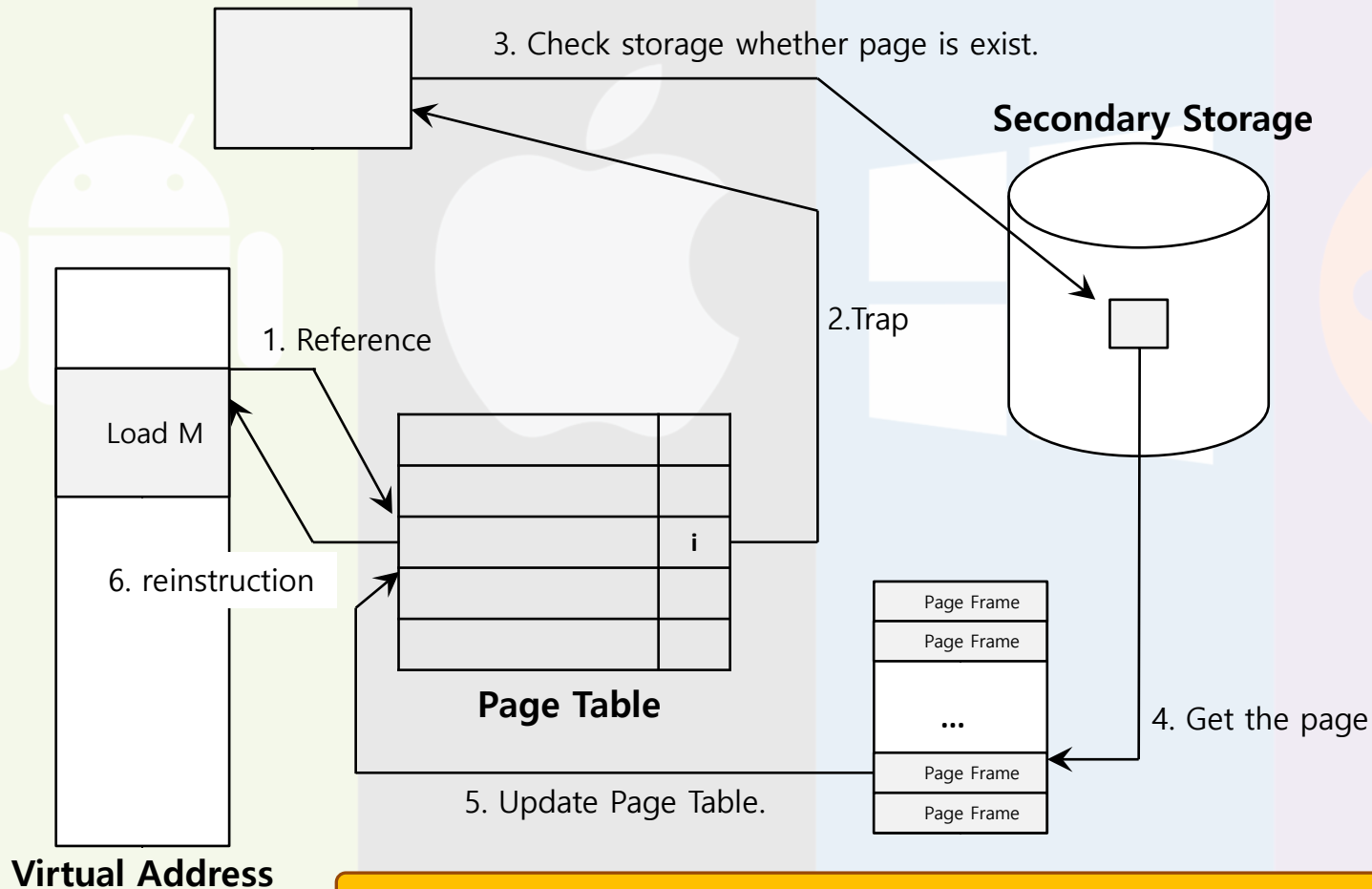
- Accessing page that is not in physical memory.

- If a page is not present and has been swapped disk, the OS need to swap the page into memory in order to service the page fault.



Page Fault Control

- PTE used for data such as the PFN of the page for a disk address



When the OS receives a page fault, it looks in the PTE and issues the request to disk.

Page Fault Control

```
1:      VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2:      (Success, TlbEntry) = TLB_Lookup(VPN)
3:      if (Success == True) // TLB Hit
4:      if (CanAccess(TlbEntry.ProtectBits) == True)
5:          Offset = VirtualAddress & OFFSET_MASK
6:          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7:          Register = AccessMemory(PhysAddr)
8:      else RaiseException(PROTECTION_FAULT)
```

Page Fault Control

```
9:      else // TLB Miss
10:      PTEAddr = PTBR + (VPN * sizeof(PTE))
11:      PTE = AccessMemory(PTEAddr)
12:      if (PTE.Valid == False)
13:          RaiseException(SEGMENTATION_FAULT)
14:      else
15:          if (CanAccess(PTE.ProtectBits) == False)
16:              RaiseException(PROTECTION_FAULT)
17:          else if (PTE.Present == True)
18:              // assuming hardware-managed TLB
19:              TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
20:              RetryInstruction()
21:          else if (PTE.Present == False)
22:              RaiseException(PAGE_FAULT)
```

Page Fault Control

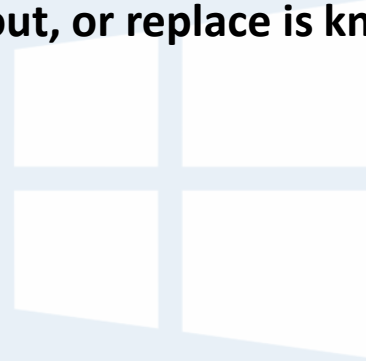
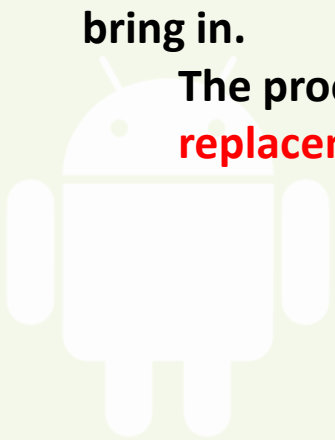
```
1:      PFN = FindFreePhysicalPage()
2:      if (PFN == -1) // no free page found
3:          PFN = EvictPage() // run replacement algorithm
4:          DiskRead(PTE.DiskAddr, pfn) // sleep (waiting for I/O)
5:          PTE.present = True // update page table with present
6:          PTE.PFN = PFN // bit and translation (PFN)
7:          RetryInstruction() // retry instruction
```

- The OS must find a physical frame for the **soon-be-faulted-in page** to reside within.
- If there is no such page, waiting for the **replacement algorithm** to run and kick some pages out of memory.

What if memory is full?

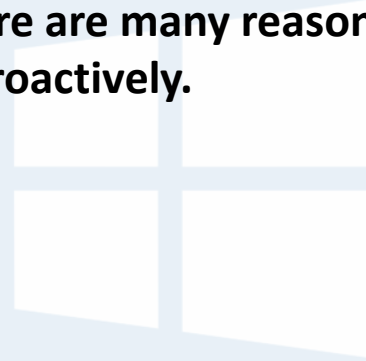
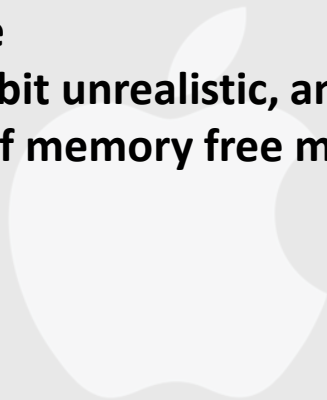
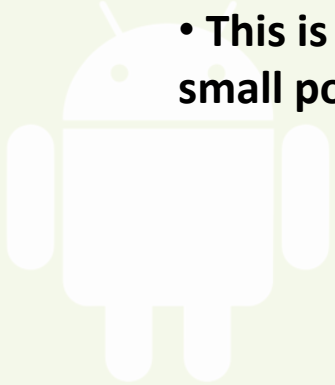
The OS likes to page out pages to make room for the new pages the OS is about to bring in.

The process of picking a page to kick out, or replace is known as **page-replacement** policy



When do page replacements actually occur?

- OS waits until memory is entirely full, and only then replaces a page to make room for some other page
 - This is a little bit unrealistic, and there are many reason for the OS to keep a small portion of memory free more proactively.



When do page replacements actually occur?

- **Swap Daemon**
 - There are fewer than “free_page_low” pages available, a background thread that is responsible for freeing memory runs.
 - The thread evicts pages until there are “free_page_high” pages available.

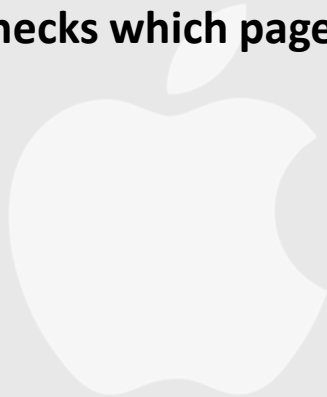


Memory is TOO Full

Memory is TOO Empty

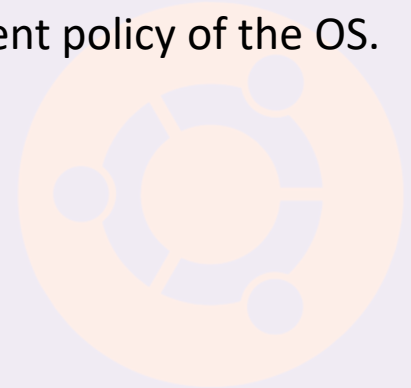
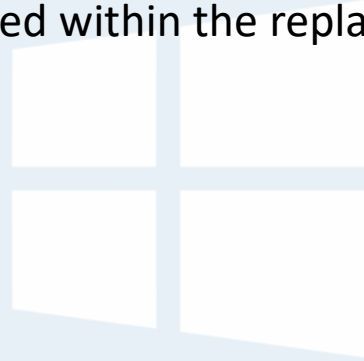
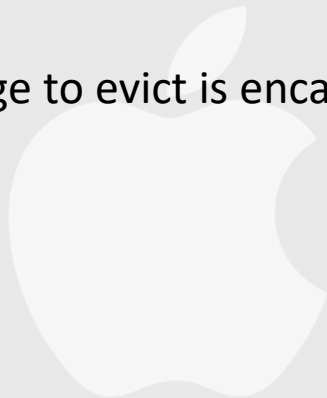
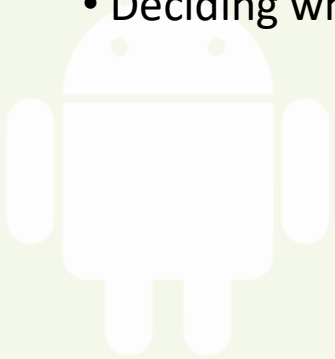
When do page replacements actually occur?

- Page Daemon
 - Periodically checks which pages can be discarded



Summary

- Memory pressure forces the OS to start paging out pages to make room for actively-used pages.
- Deciding which page to evict is encapsulated within the replacement policy of the OS.



Cache Management

- Goal in picking a replacement policy for this cache is to minimize the number of cache misses



Cache Management

- Goal in picking a replacement policy for this cache is to minimize the number of cache misses.
- The number of cache hits and misses let us calculate the average memory access time(AMAT).

Argument	Meaning
T_M	The cost of accessing memory
T_D	The cost of accessing disk
P_{hit}	The probability of finding the data item in the cache(a hit)
P_{miss}	The probability of not finding the data in the cache(a miss)

$$AMAT = T_M + (P_{miss} \times T_D)$$

Cache Management - Example

- For example, let us imagine a machine with a (tiny) address space: 4KB, with 256-byte pages.
- Thus, a virtual address has two components: a 4-bit VPN (the most-significant bits) and an 8-bit offset (the least-significant bits).
- Thus, a process in this example can access 16 total virtual pages.

Cache Management - Example

- address space: 4KB, with 256-byte pages.
- 4-bit VPN, 8-bit offset
- 16 total virtual pages.
- In this example, the process generates the following memory references (i.e., virtual addresses): 0x000, 0x100, 0x200, 0x300, 0x400, 0x500, 0x600, 0x700, 0x800, 0x900.
- These virtual addresses refer to the first byte of each of the first ten pages of the address space (the page number being the first hex digit of each virtual address).
- Consider 0x800 the first hex digit is $8_{16} = 1000_2$ Therefore the VPN for 0x800 = 1000_2

Cache Management - Example

- address space: 4KB, with 256-byte pages.
 - 4-bit VPN, 8-bit offset
 - 16 total virtual pages.
 - (i.e., virtual addresses): 0x000, 0x100, 0x200, 0x300, 0x400, 0x500, 0x600, 0x700, 0x800, 0x900.
 - Consider 0x800 the first hex digit is $8_{16} = 1000_2$ Therefore the VPN for 0x800 = 1000_2
 - **Let us further assume that every page except virtual page 3 is already in memory.**
- Thus, our sequence of memory references will encounter the following behaviour:**
hit, hit, hit, miss, hit, hit, hit, hit, hit, hit.

Cache Management - Example

- address space: 4KB, with 256-byte pages.
- 4-bit VPN, 8-bit offset
- 16 total virtual pages.
- (i.e., virtual addresses): 0x000, 0x100, 0x200, 0x300, 0x400, 0x500, 0x600, 0x700, 0x800, 0x900.
- Consider 0x800 the first hex digit is $8_{16} = 1000_2$ Therefore the VPN for 0x800 = 1000_2
- hit, hit, hit, miss, hit, hit, hit, hit, hit, hit.
- **We can compute the hit rate (the percent of references found in memory): 90%, as 9 out of 10 references are in memory. The miss rate is thus 10% (PMiss = 0.1). In general, PHit + PMiss = 1.0**

Cache Management - Example

- address space: 4KB, with 256-byte pages.
- 4-bit VPN, 8-bit offset
- 16 total virtual pages.
- (i.e., virtual addresses): 0x000, 0x100, 0x200, 0x300, 0x400, 0x500, 0x600, 0x700, 0x800, 0x900.
- Consider 0x800 the first hex digit is $8_{16} = 1000_2$ Therefore the VPN for 0x800 = 1000_2
- hit, hit, hit, miss, hit, hit, hit, hit, hit, hit.
- (Phit = 0.9), (PMiss = 0.1)
- **To calculate AMAT, we need to know the cost of accessing memory and the cost of accessing disk.**
- **Assuming the cost of accessing memory (TM) is around 100 nanoseconds, and the cost of accessing disk (TD) is about 10 milliseconds**

Cache Management - Example

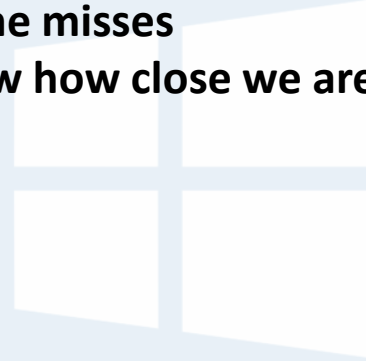
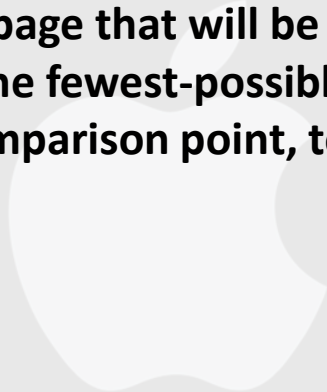
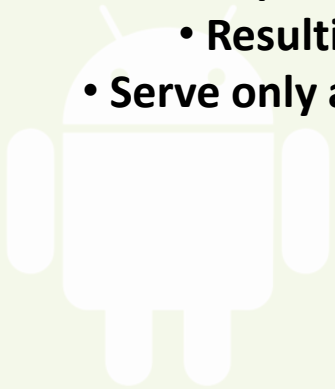
- address space: 4KB, with 256-byte pages.
- 4-bit VPN, 8-bit offset
- 16 total virtual pages.
- (i.e., virtual addresses): 0x000, 0x100, 0x200, 0x300, 0x400, 0x500, 0x600, 0x700, 0x800, 0x900.
- Consider 0x800 the first hex digit is $8_{16} = 1000_2$ Therefore the VPN for 0x800 = 1000_2
- hit, hit, hit, miss, hit, hit, hit, hit, hit.
- (Phit = 0.9), (PMiss = 0.1)
- (TM = 100 nanoseconds), (TD = 10 milliseconds)
- **We have the following AMAT: $100\text{ns} + 0.1 \cdot 10\text{ms}$, which is $100\text{ns} + 1\text{ms}$, or 1.0001 ms, or about 1 millisecond.**
- **1Microsecond = 1000nano**
- **$100\text{ns} + 0.1 \cdot 10(1000) = 1100$**

Cache Management - Example

- address space: 4KB, with 256-byte pages.
- 4-bit VPN, 8-bit offset
- 16 total virtual pages.
- (i.e., virtual addresses): 0x000, 0x100, 0x200, 0x300, 0x400, 0x500, 0x600, 0x700, 0x800, 0x900.
- Consider 0x800 the first hex digit is $8_{16} = 1000_2$ Therefore the VPN for 0x800 = 1000_2
- hit, hit, hit, miss, hit, hit, hit, hit, hit, hit.
- (Phit = 0.9), (PMiss = 0.1)
- (TM = 100 nanoseconds), (TD = 10 milliseconds)
- We have the following AMAT: $100\text{ns} + 0.1 \cdot 10\text{ms}$, which is $100\text{ns} + 1\text{ms}$, or 1.0001 ms, or about 1 millisecond.
- **If our hit rate had instead been 99.9% (Pmiss = 0.001), the result is quite different: AMAT is 10.1 microseconds, or roughly 100 times faster.**
- **As the hit rate approaches 100%, AMAT approaches 100 nanoseconds.**

The Optimal Replacement Policy

- Leads to the fewest number of misses overall
 - Replaces the page that will be accessed furthest in the future
 - Resulting in the fewest-possible cache misses
- Serve only as a comparison point, to know how close we are to perfect



Tracing the Optimal Policy

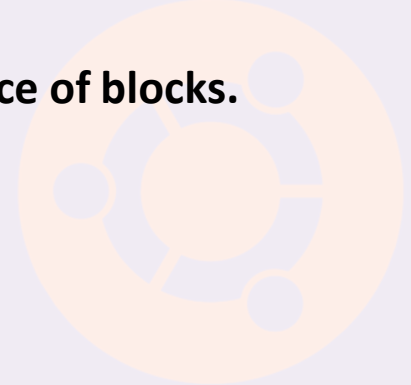
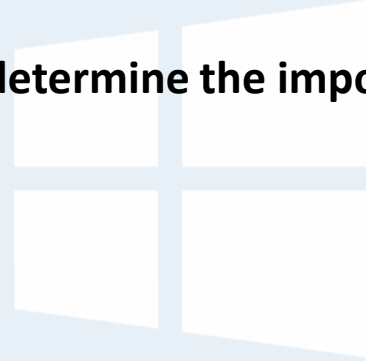
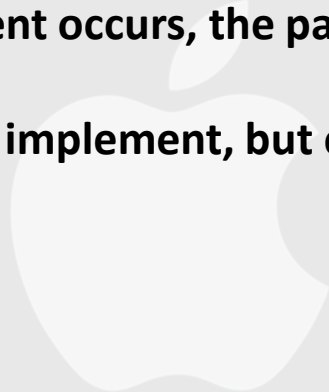
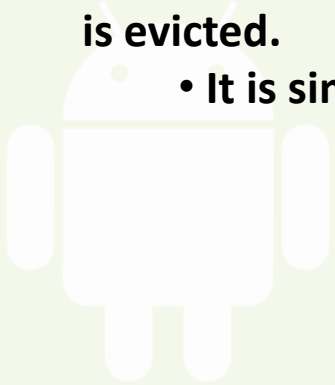
Reference Row

0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	2	0,1,3
0	Hit		0,1,3
3	Hit		0,1,3
1	Hit		0,1,3
2	Miss	3	0,1,2
1	Hit		0,1,2

A Simple Policy: FIFO

- Pages were placed in a queue when they enter the system.
- When a replacement occurs, the page on the tail of the queue(the “First-in” pages) is evicted.
 - It is simple to implement, but can't determine the importance of blocks.



Tracing FIFO

Reference Row

0 1 2 0 1 3 0 3 1 2 1

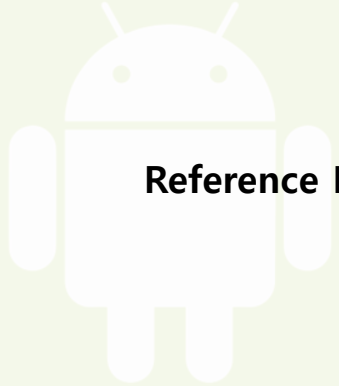
Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	0	1,2,3
0	Miss	1	2,3,0
3	Hit		2,3,0
1	Miss		3,0,1
2	Miss	3	0,1,2
1	Hit		0,1,2

Hit rate is $\frac{\text{Hits}}{\text{Hits} + \text{Misses}} = 36.4\%$

Even though page 0 had been accessed a number of times, **FIFO still kicks it out.**

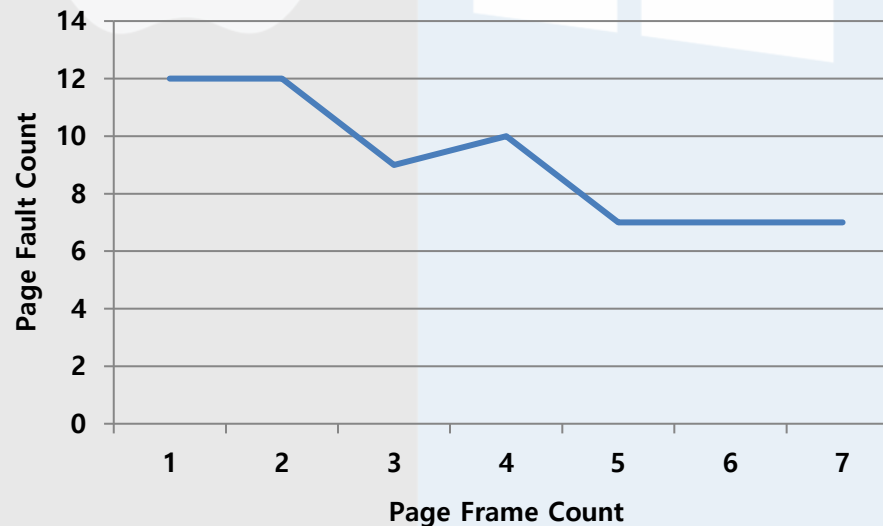
BELADY'S ANOMALY

- We would expect the cache hit rate to increase when the cache gets larger. But in this case, with FIFO, something strange occurs.



Reference Row

1 2 3 4 1 2 5 1 2 3 4 5



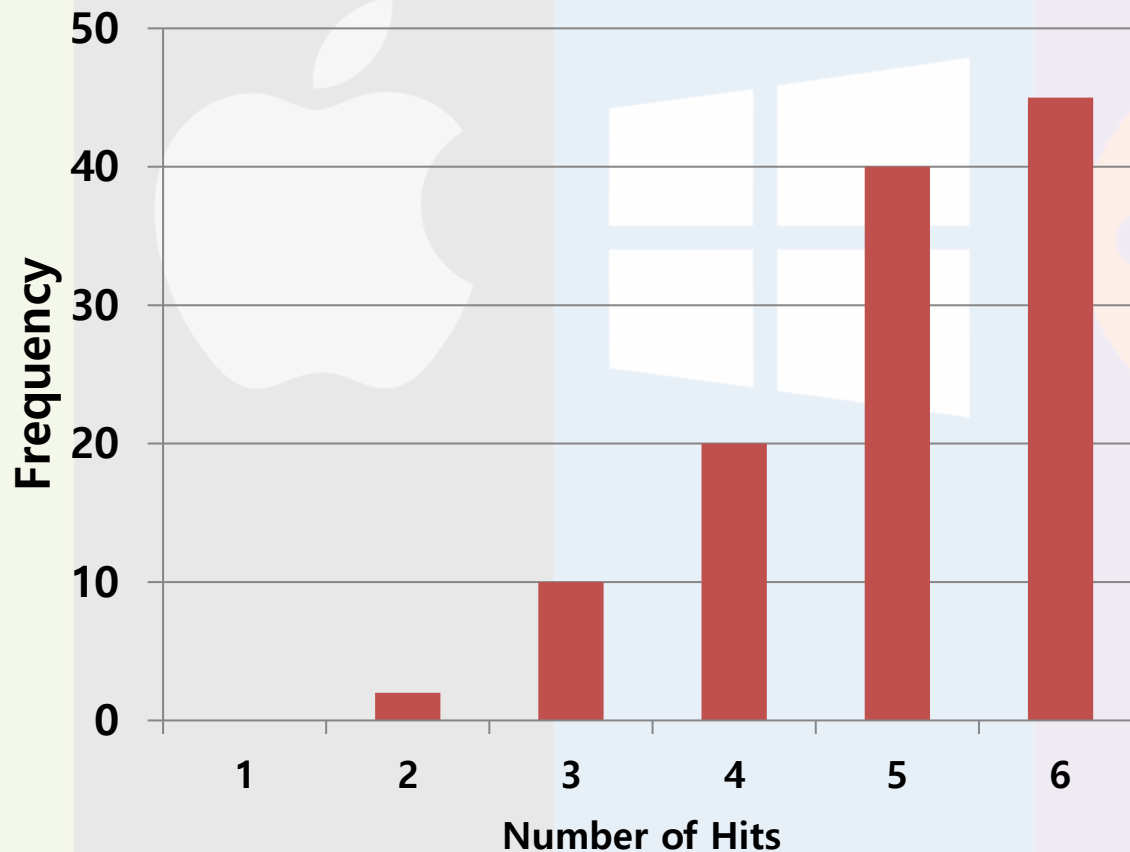
A Simple Policy: Random

- Picks a random page to replace under memory pressure.
 - It doesn't really try to be too intelligent in picking which blocks to evict.
 - Random does depends entirely upon how lucky Random gets in its choice

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	0	1,2,3
0	Miss	1	2,3,0
3	Hit		2,3,0
1	Miss	3	2,0,1
2	Hit		2,0,1
1	Hit		2,0,1

A Simple Policy: Random

- Sometimes, Random is as good as optimal, achieving 6 hits on the example trace.



Random Performance over 10,000 Trials

Using History

- Lean on the past and use history.
 - Two type of historical information.

Historical Information	Meaning	Algorithms
recency	The more recently a page has been accessed, the more likely it will be accessed again	LRU
frequency	If a page has been accessed many times, It should not be replaced as it clearly has some value	LFU

Using History - LRU

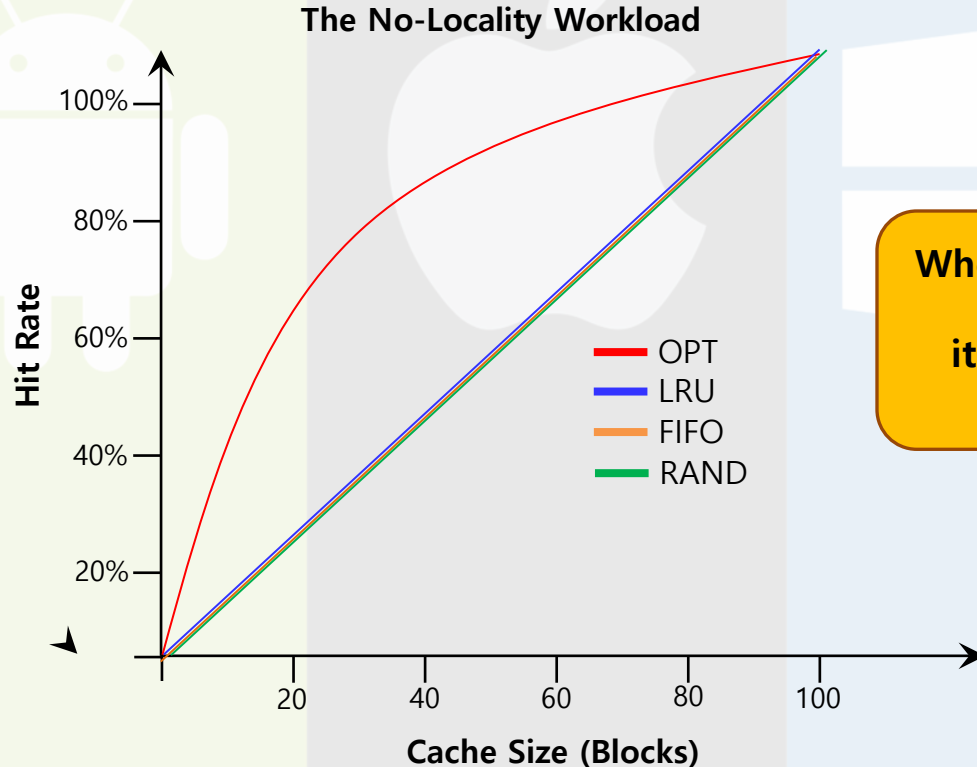
Reference Row

0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		1,2,0
1	Hit		2,0,1
3	Miss	2	0,1,3
0	Hit		1,3,0
3	Hit		1,0,3
1	Hit		0,3,1
2	Miss	0	3,1,2
1	Hit		3,2,1

Workload Example : The No-Locality Workload

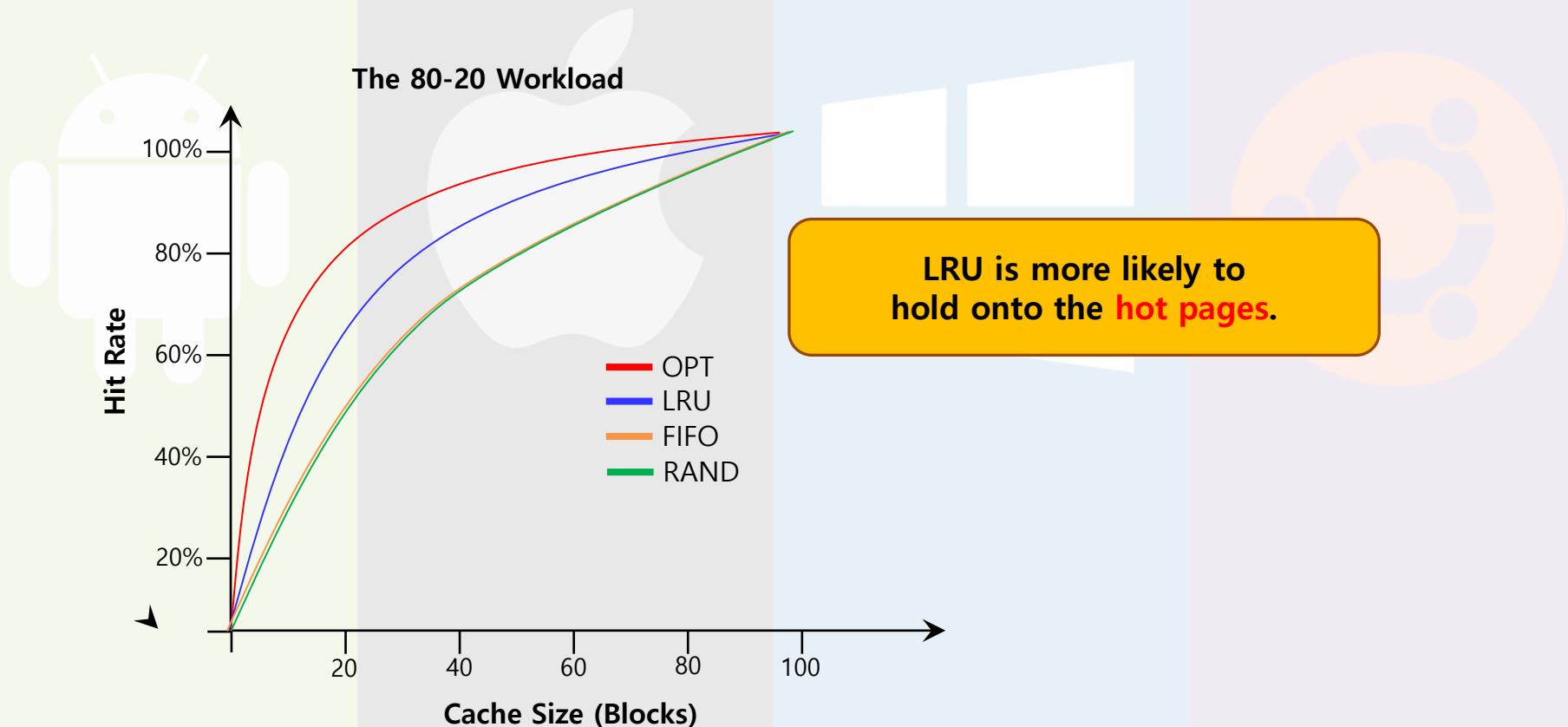
- Each reference is to a random page within the set of accessed pages.
 - Workload accesses 100 unique pages over time.
 - Choosing the next page to refer to at random



When the cache is large enough to fit the entire workload, it also **doesn't matter** which policy you use.

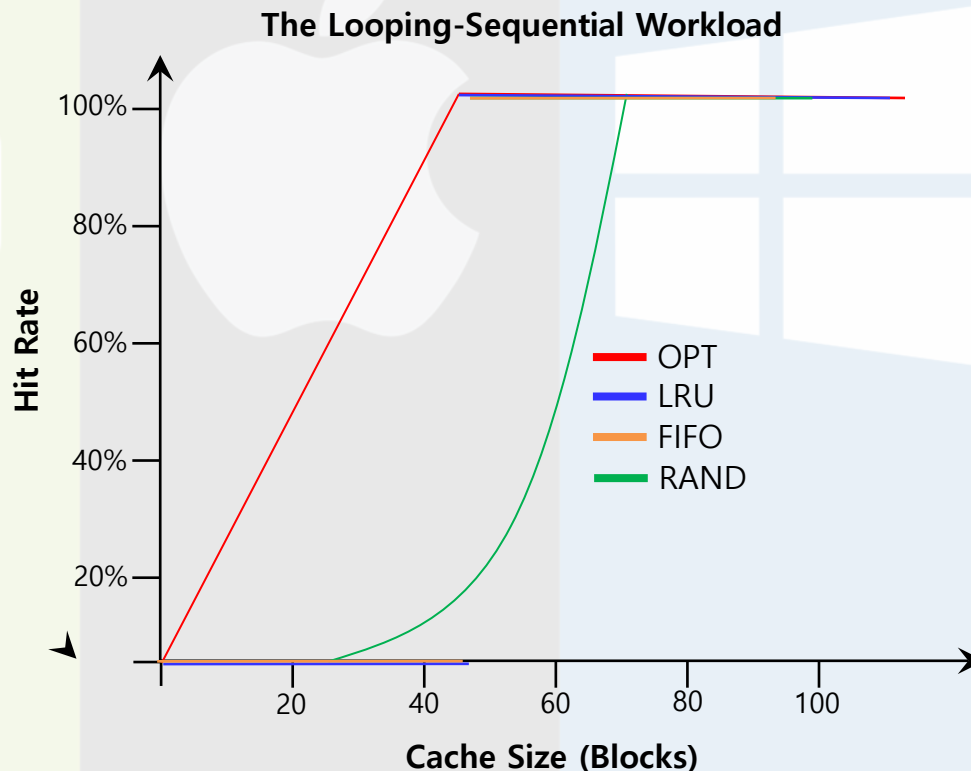
Workload Example : The 80-20 Workload

- Exhibits locality: 80% of the reference are made to 20% of the page
- The remaining 20% of the reference are made to the remaining 80% of the pages.



Workload Example : The Looping Sequential

- Refer to 50 pages in sequence.
 - Starting at 0, then 1, ... up to page 49, and then we Loop, repeating those accesses, for total of 10,000 accesses to 50 unique pages.



Implementing Historical Algorithms

- To keep track of which pages have been least-and-recently used, the system has to do some accounting work on every memory reference.
- Add a little bit of hardware support.



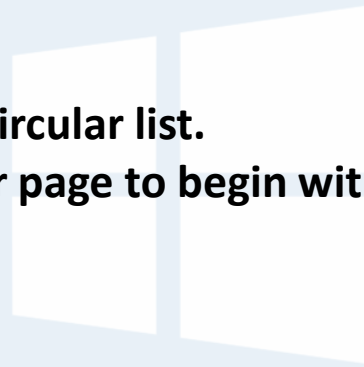
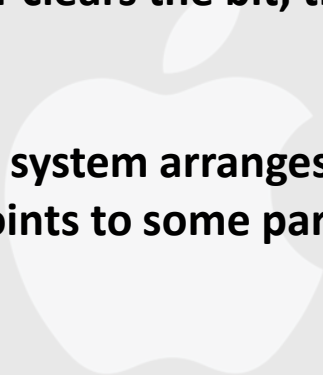
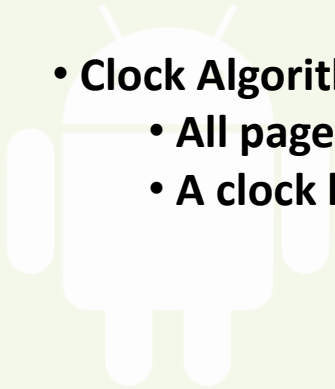
Approximating LRU

- Require some hardware support, in the form of a use bit
 - Whenever a page is referenced, the use bit is set by hardware to 1.
 - Hardware never clears the bit, though; that is the responsibility of the OS



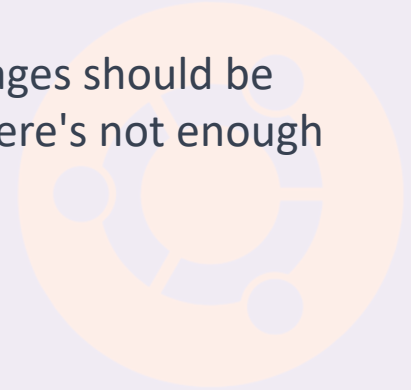
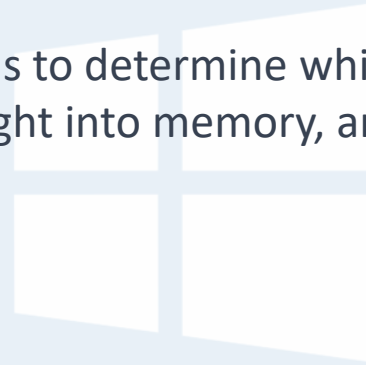
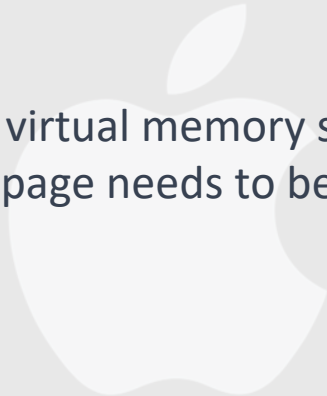
Approximating LRU

- Require some hardware support, in the form of a use bit
 - Whenever a page is referenced, the use bit is set by hardware to 1.
 - Hardware never clears the bit, though; that is the responsibility of the OS
- Clock Algorithm
 - All pages of the system arranges in a circular list.
 - A clock hand points to some particular page to begin with.



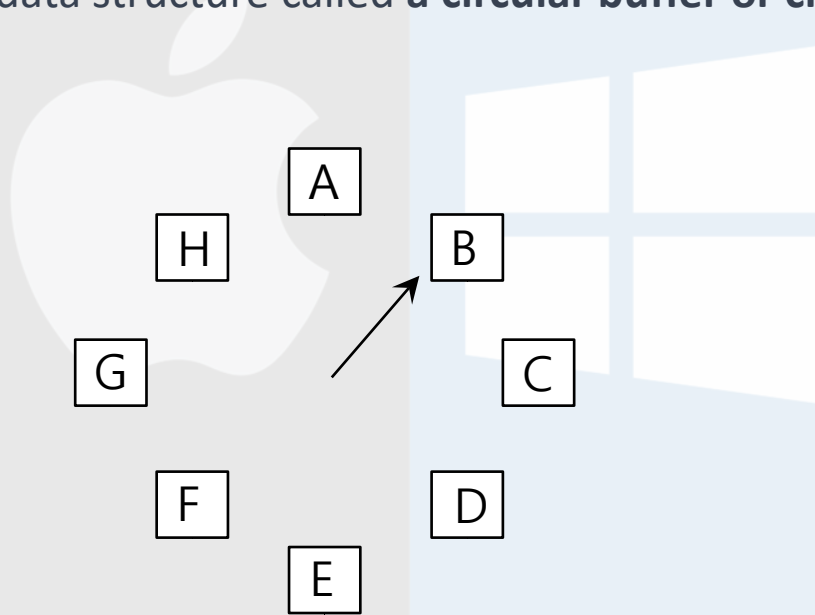
Clock Algorithm

- The "**Clock**" algorithm, also known as the "**Second-Chance**" algorithm, is a **page replacement policy** used in operating systems to manage memory and control swapping.
- It's primarily used in virtual memory systems to determine which pages should be replaced when a new page needs to be brought into memory, and there's not enough free space available.



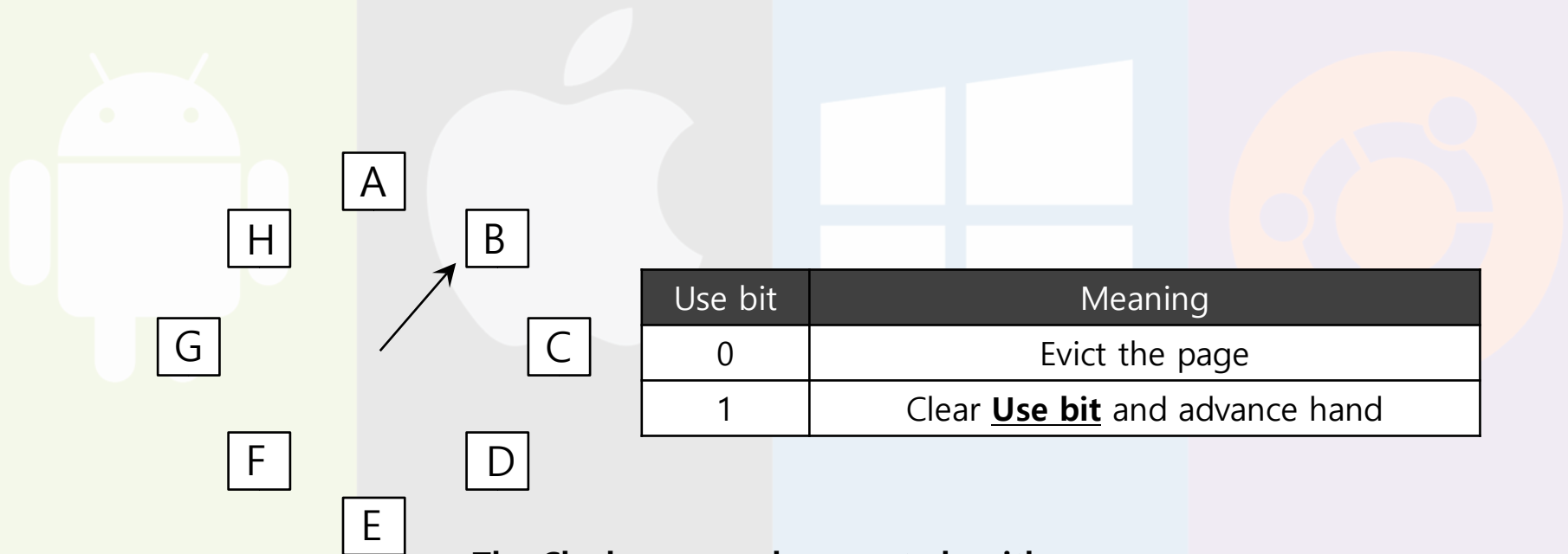
Clock Algorithm

- The basic idea behind the Clock algorithm is to maintain a **circular list** of pages in memory, similar to the way a clock's hands move around its face. This circular list is implemented using a data structure called a **circular buffer** or **circular linked list**



Clock Algorithm

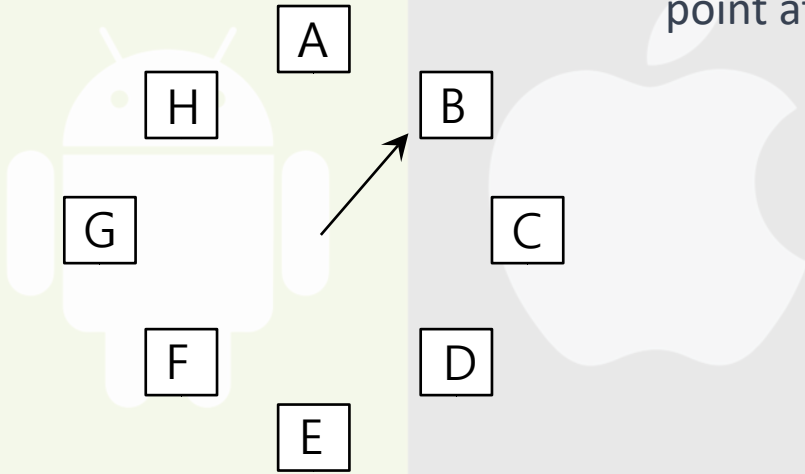
- Each page in the list is **associated** with a “**Use bit**” that indicates whether the page has been accessed (read or written to).



The Clock page replacement algorithm

Clock Algorithm

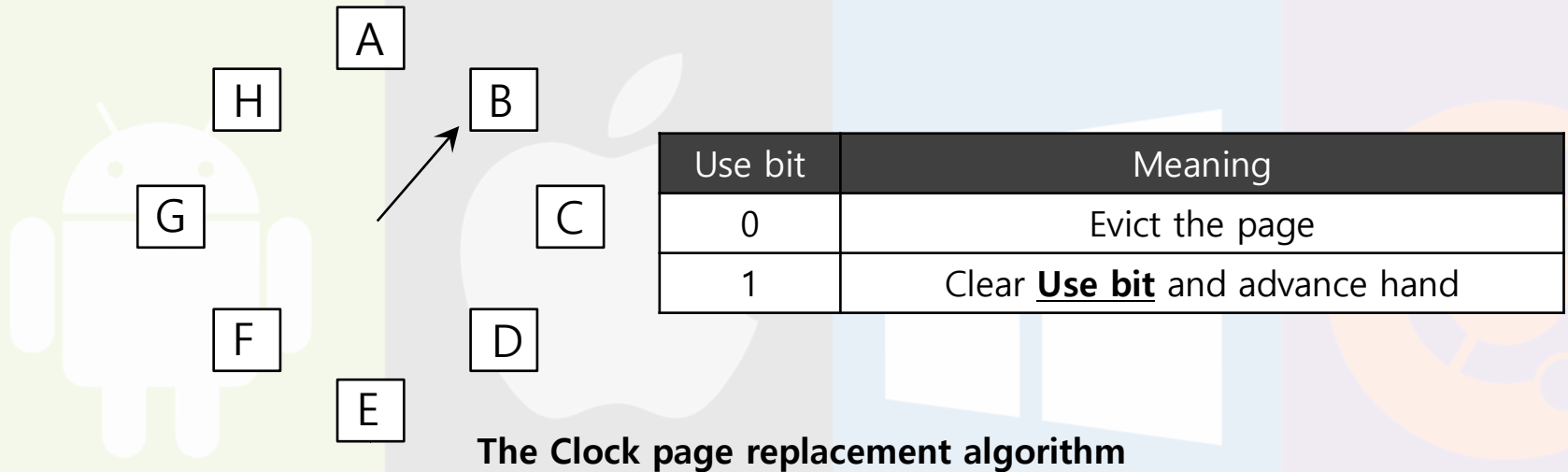
1. Initialization: Each page frame in memory is represented in the circular list. The list is initialized to point at a random frame.



Use bit	Meaning
0	Evict the page
1	Clear Use bit and advance hand

Clock Algorithm

- The algorithm continues until it finds a use bit that is set to 0.

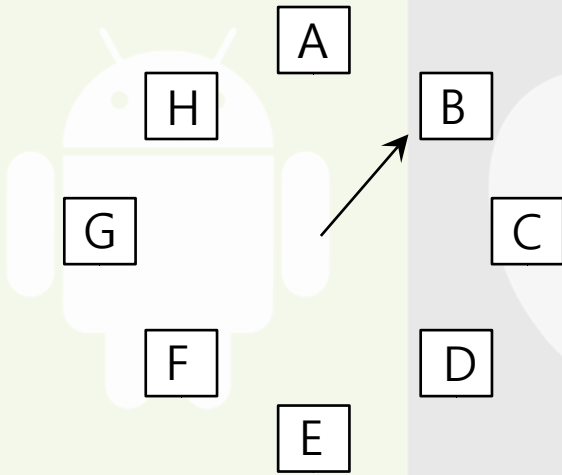


When a page fault occurs, the page the hand is pointing to is inspected.
The action taken depends on the Use bit

Clock Algorithm

1.Initialization: Each page frame in memory is represented in the circular list. The list is initialized to point at a random frame.

2.Page Request: When a new page needs to be brought into memory, the operating system checks the page frames in the circular list in the order they appear. It starts at the current position and proceeds in a circular manner.



Use bit	Meaning
0	Evict the page
1	Clear Use bit and advance hand

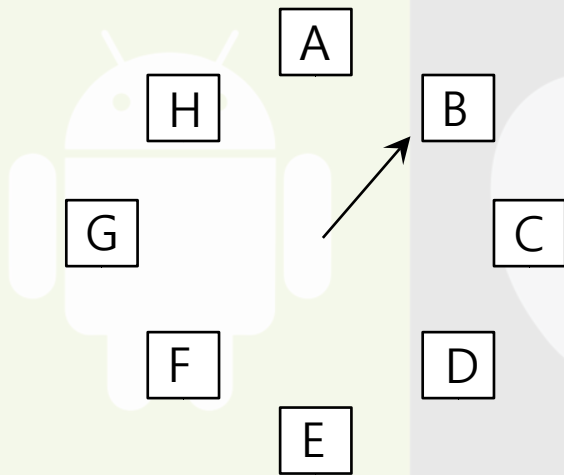
Clock Algorithm

1.Initialization: Each page frame in memory is represented in the circular list. The list is initialized to point at a random frame.

2.Page Request: When a new page needs to be brought into memory, the operating system checks the page frames in the circular list in the order they appear. It starts at the current position and proceeds in a circular manner.

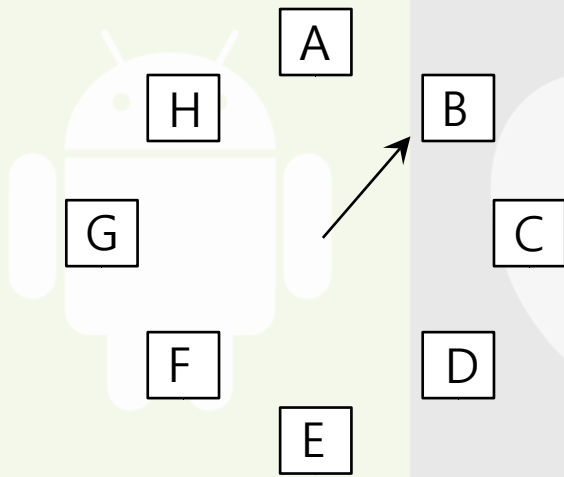
3.Page Selection: For each page frame visited, the reference bit of that frame is checked:

1. If the reference bit is 0, this page frame is a candidate for replacement. The new page is loaded into this frame, and the current position in the circular list is advanced to the next frame.
2. If the reference bit is 1, it means the page has been recently accessed. The reference bit is cleared to 0, indicating that the page hasn't been used in the current examination cycle. The current position is then advanced to the next frame, and the search continues.



Use bit	Meaning
0	Evict the page
1	Clear Use bit and advance hand

Clock Algorithm



1.Initialization: Each page frame in memory is represented in the circular list. The list is initialized to point at a random frame.

2.Page Request: When a new page needs to be brought into memory, the operating system checks the page frames in the circular list in the order they appear. It starts at the current position and proceeds in a circular manner.

3.Page Selection: For each page frame visited, the reference bit of that frame is checked:

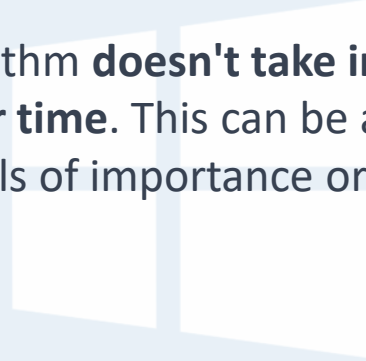
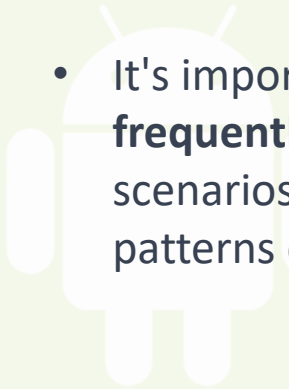
1. If the reference bit is 0, this page frame is a candidate for replacement. The new page is loaded into this frame, and the current position in the circular list is advanced to the next frame.
2. If the reference bit is 1, it means the page has been recently accessed. The reference bit is cleared to 0, indicating that the page hasn't been used in the current examination cycle. The current position is then advanced to the next frame, and the search continues.

4.Repeat: Steps 2 and 3 are repeated until a suitable page frame for replacement is found. This might involve multiple passes through the circular list.

Use bit	Meaning
0	Evict the page
1	Clear Use bit and advance hand

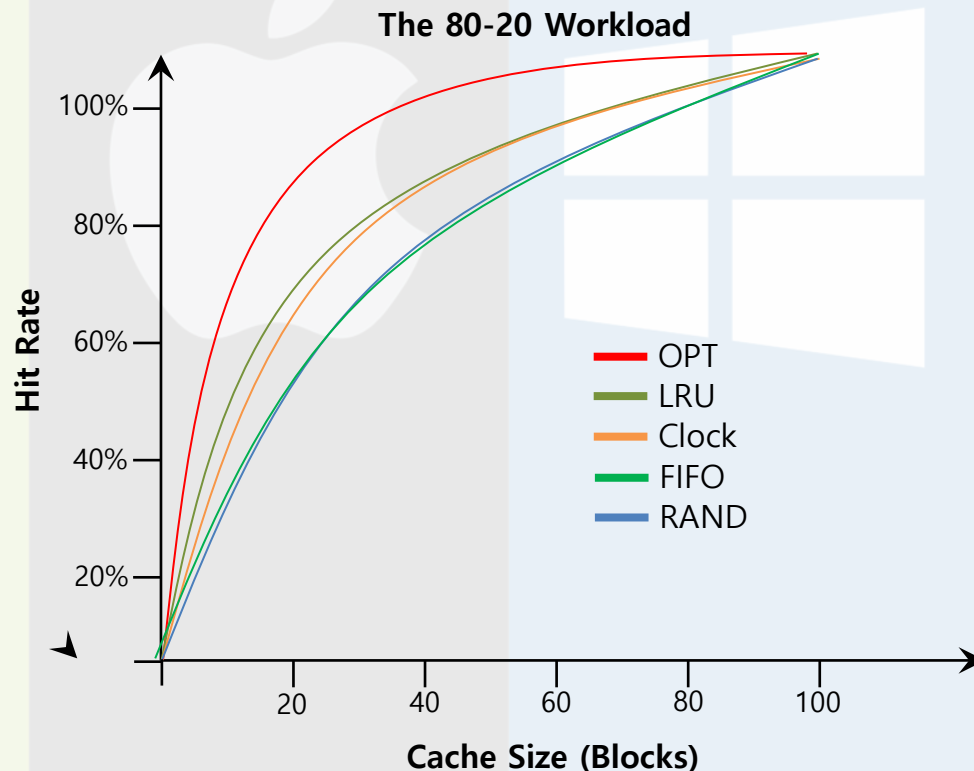
Clock Algorithm Summary

- By giving pages a "second chance" before they are considered for replacement, the Clock algorithm aims to **prioritize retaining pages that are actively used** while still allowing pages that are not being actively used to eventually be replaced.
- It's important to note that the Clock algorithm **doesn't take into account how frequently a page has been accessed over time**. This can be a limitation in scenarios where pages have different levels of importance or where access patterns change dynamically.



Workload Example : Clock Algorithm

- Clock algorithm doesn't do as well as perfect LRU, it does better than approaches that don't consider history at all.



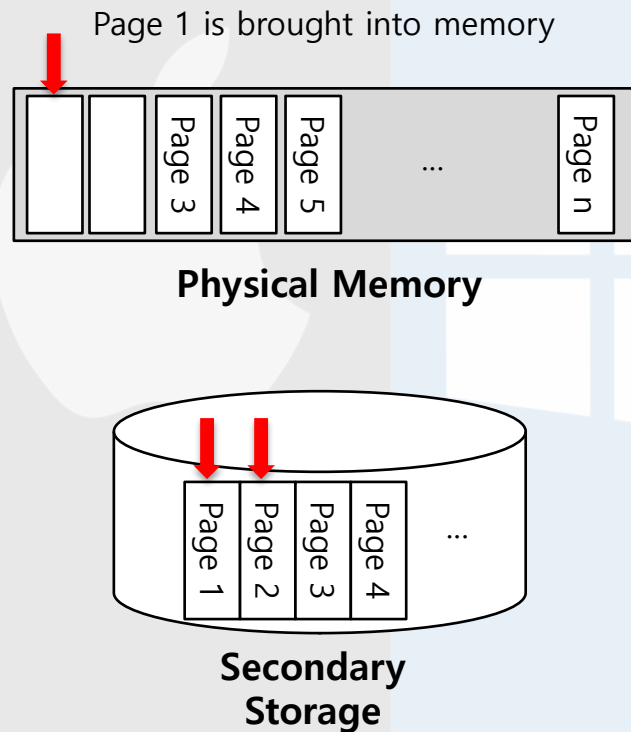
Page Selection Policy

- The OS has to decide when to bring a page into memory.
- Presents the OS with some different options.



Prefetching

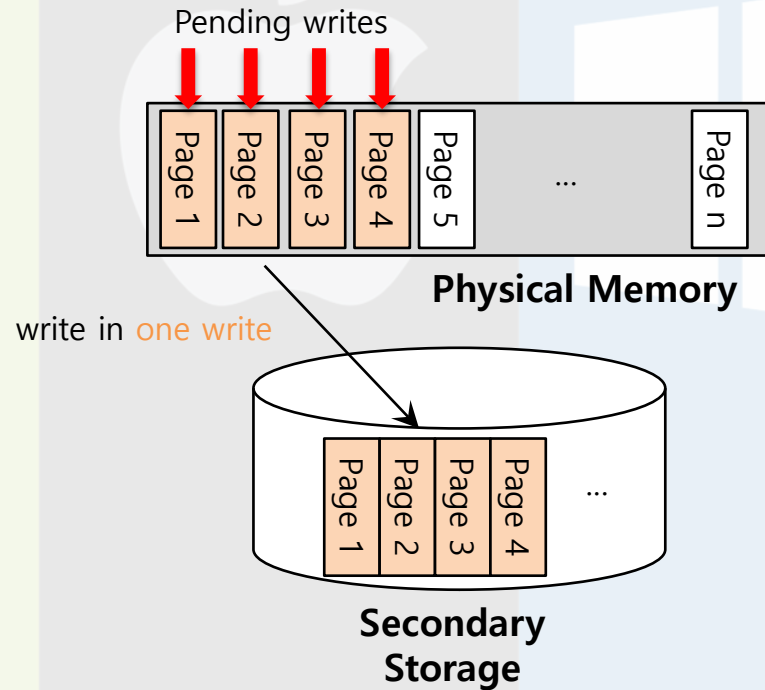
- The OS guess that a page is about to be used, and thus bring it in ahead of time.



Page 2 likely soon be accessed and thus should be brought into memory too

Clustering, Grouping

- Collect a number of pending writes together in memory and write them to disk in one write.
- Perform a single large write more efficiently than many small ones.



Thrashing

- Memory is oversubscribed and the memory demands of the set of running processes exceeds the available physical memory.
- Leads to a constant state of paging and page faults, inhibiting most application-level processing

