Swapping (ch. 21 + 22)

Operating Systems

Based on: Three Easy Pieces by Arpaci-Dusseaux

Moshe Sulamy

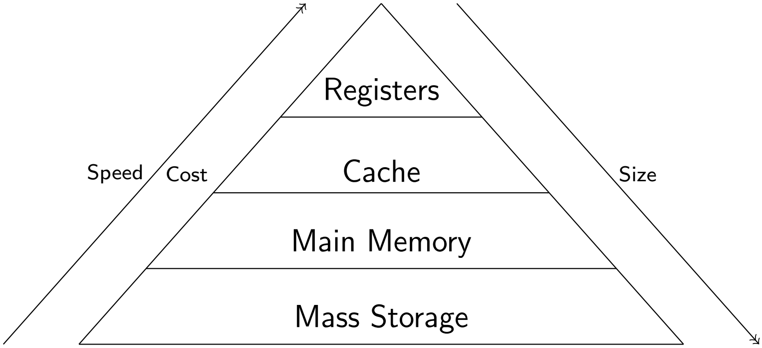
Tel-Aviv Academic College

Swapping

- Process **address space**: 4GB
  - Ten processes: 40GB
  - Usually much more
- To support large address spaces:
  - Additional level in the **memory hierarchy**
  - Store pages that aren't in demand to **hard disk drive**

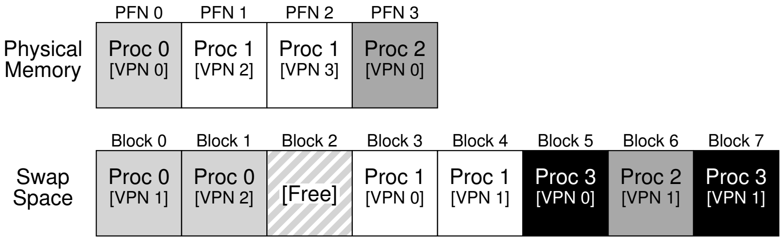
How can we use larger, slower devices to virtualize a large address space?

Memory Hierarchy

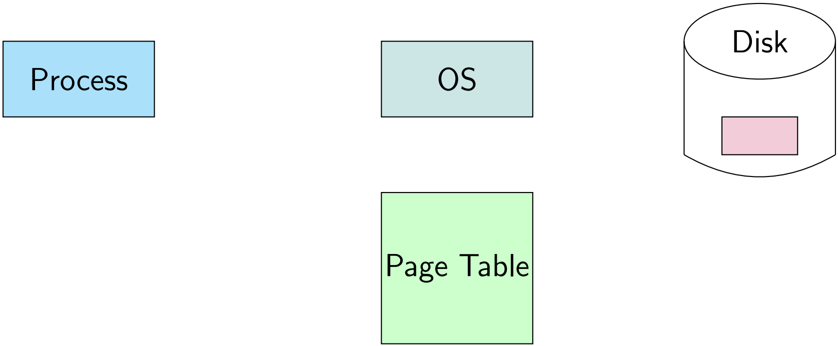


Swap Space

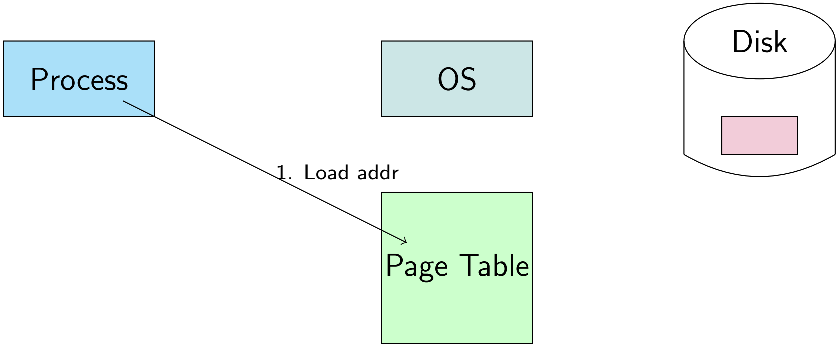
- Reserve space on disk for moving pages back and forth
- **Swap space**
  - We **swap** pages out of memory to it, into memory from it



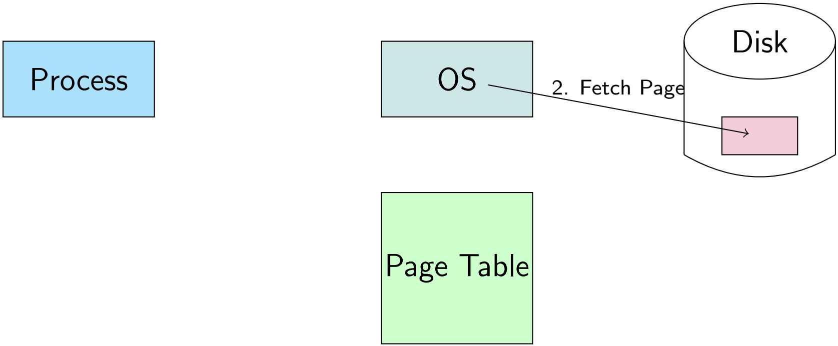
Page Fault



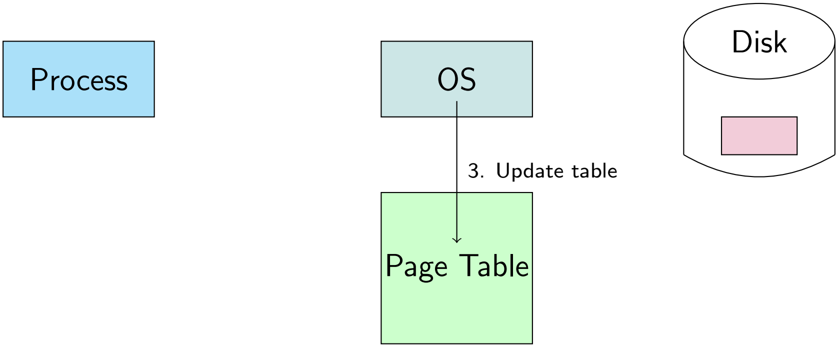
Page Fault



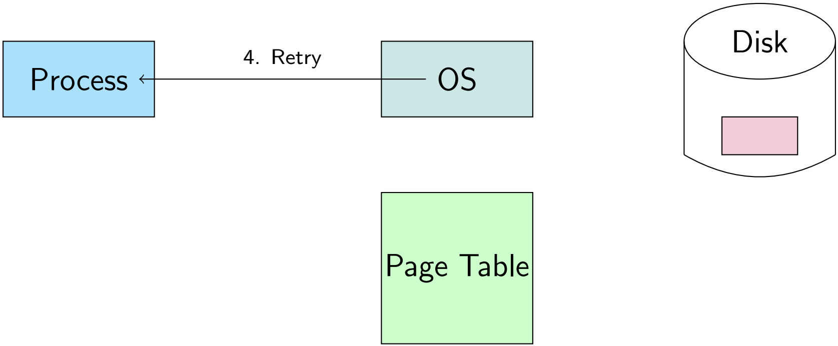
Page Fault



Page Fault



Page Fault



Page Fault

- If a page is not present, **page-fault handler** runs
  - Handled in software
  - See where (and if!) the page exists.
  - If page is nowhere, process terminates.
  - If page in memory fix the page table and retry.
  - If page in disk then read page from disk to memory
  - Process **blocked** until done
- When the I/O completes:
  - Update page table (mark as present)
  - retry the instruction

What If Memory Is Full?

- First **page out** a page, to **page in** from swap space
- Picking a page to evict: **page-replacement policy**

How can the OS decide which page to evict from memory?

When Replacements Really Occur

- OS doesn't wait until memory is full
- **Swap daemon** (or **page daemon**)
  - Background process
  - Fewer than LW (**low-watermark**) pages: free memory
  - Evicts pages until there are HW (**high-watermark**) available

Cache Management

- Minimize number of **cache misses**
- **Average memory access time (AMAT)**:

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

- T_M: cost of accessing memory
- P_{miss}: probability of cache miss
- T_D: cost of accessing disk

Optimal Replacement Policy

- Leads to fewest number of misses
- Replace page accessed furthest in the future
- Can't build optimal policy
  - Used only as a comparison point

Optimal Replacement Policy

Access	Hit/Miss	Evict	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

Optimal Replacement Policy

Access	Hit/Miss	Evict	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

- Hit rate: $\frac{6}{6+5} =$ **54.5%**

Optimal Replacement Policy

Access	Hit/Miss	Evict	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

- Hit rate: $\frac{6}{6+5} = \mathbf{54.5\%}$
- Without **cold-start**: **85.7%**

FIFO

- Place pages in a queue
- Evict page on the tail of the queue

FIFO

Access	Hit/Miss	Evict	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	2	3,0,1
2	Miss	3	0,1,2
1	Hit		0,1,2

- Hit rate:

FIFO

Access	Hit/Miss	Evict	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	2	3,0,1
2	Miss	3	0,1,2
1	Hit		0,1,2

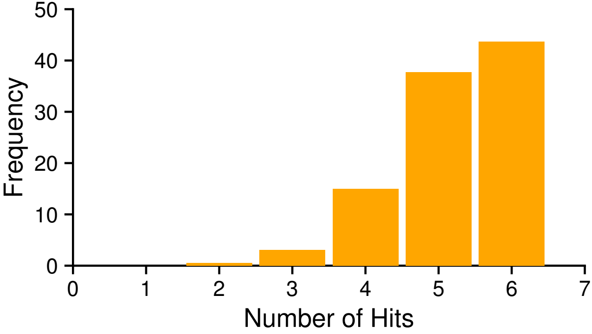
- Hit rate: 36.4% (or 57.1%)

Belady's Anomaly

- We expect hit rate to improve as cache gets larger
  - With FIFO, it gets worse!
- Policies such as LRU have a **stack property**
  - Cache of size $N+1$ includes contents of cache of size N
  - Thus, increasing cache size can't harm hit rate

Random

- Pick a random page to replace
  - Performance is... random



Using History

- Use historical information
  - **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, clearly it has some value (**LFU**)

LRU

Access	Hit/Miss	Evict	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

- Hit rate?

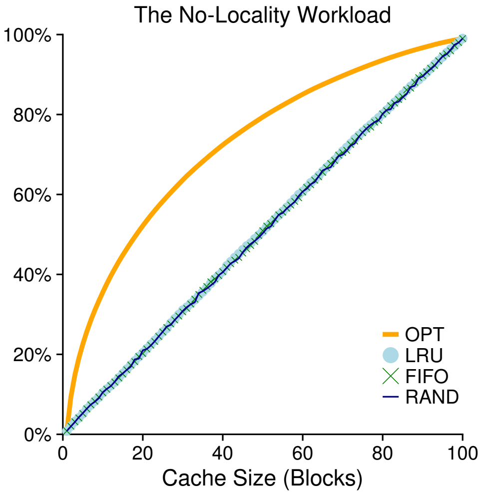
LRU

Access	Hit/Miss	Evict	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

- Hit rate? **optimal** (in this example)

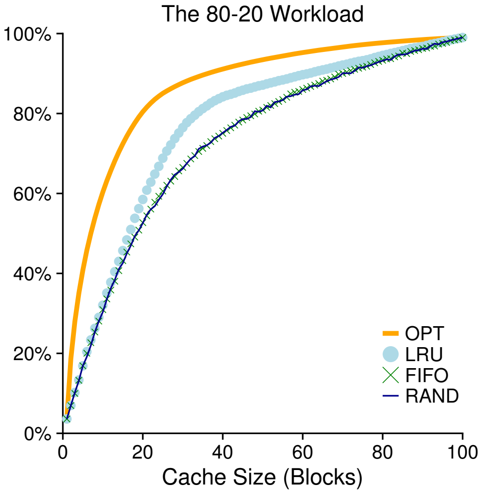
Workload Examples

- **No-Locality**: 10,000 random accesses to 100 unique pages



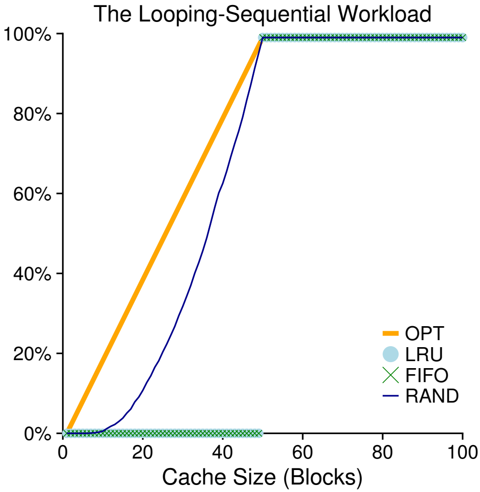
Workload Examples

- **80-20**: 80% of references to 20% of pages



Workload Examples

- **Looping**: Access 50 pages in a loop

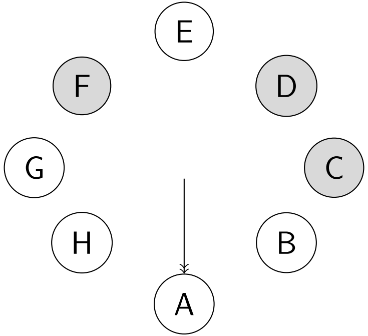


Implementing Historical Algorithms

- To keep track of LRU, update <u>on every memory reference</u>
  - Hardware support: time field
- Finding LRU is expensive
  - Scanning 1 million pages
  - Do we really need the absolute oldest?

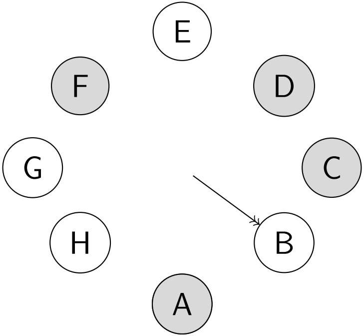
Approximating LRU

- **Use bit**
  - Set by hardware whenever a page is referenced
- **Clock algorithm**
  - Arrange pages in a circular list (i.e., clock)
  - Clock hand points to some page
  - Cycle clock until **use bit**=0



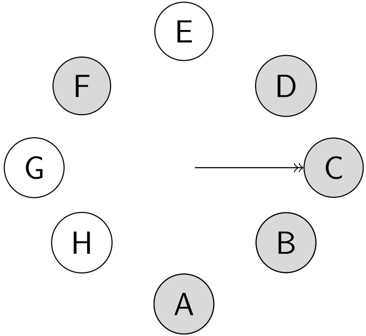
Approximating LRU

- **Use bit**
  - Set by hardware whenever a page is referenced
- **Clock algorithm**
  - Arrange pages in a circular list (i.e., clock)
  - Clock hand points to some page
  - Cycle clock until **use bit**=0



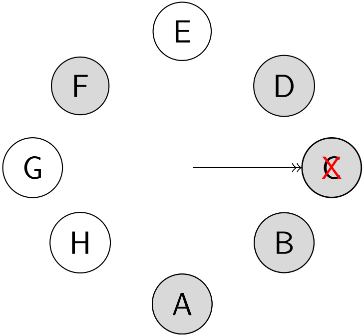
Approximating LRU

- **Use bit**
  - Set by hardware whenever a page is referenced
- **Clock algorithm**
  - Arrange pages in a circular list (i.e., clock)
  - Clock hand points to some page
  - Cycle clock until **use bit**=0



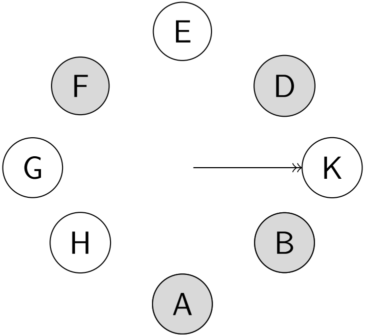
Approximating LRU

- **Use bit**
  - Set by hardware whenever a page is referenced
- **Clock algorithm**
  - Arrange pages in a circular list (i.e., clock)
  - Clock hand points to some page
  - Cycle clock until **use bit**=0



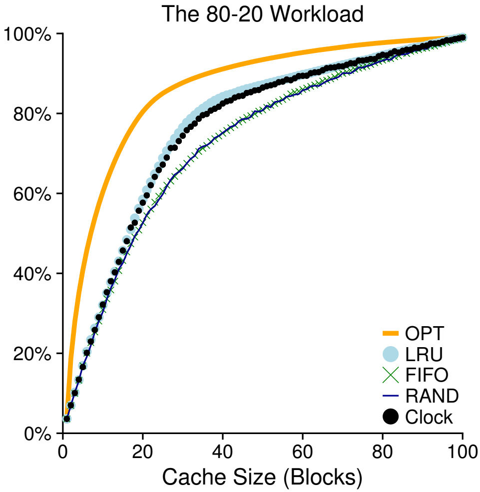
Approximating LRU

- **Use bit**
  - Set by hardware whenever a page is referenced
- **Clock algorithm**
  - Arrange pages in a circular list (i.e., clock)
  - Clock hand points to some page
  - Cycle clock until **use bit**=0



Approximating LRU

- 80-20 with clock



Considering Dirty Pages

- **Modified bit (dirty bit)**
  - If a page has been **modified** it must be written to disk to evict
  - If it is **clean**, eviction is free

  - Clock can be modified: first look for unused and clean

VM Policies

- **Prefetching**: common policy of **demand paging**
  - If code page P is brought into memory
  - Code page $P + 1$ will likely soon be accessed
- **Clustering** (**grouping**) of writes
  - Collect a number of pending writes together
  - Effective due to the nature of disk drives

Summary

- **Swapping**: store pages in hard disk drive
  - **Present bit**: is page in physical memory
- **Page replacement policy**
  - Minimize number of **cache misses**
  - Far from optimal: FIFO, random
- **LRU**
  - Evict least-recently used
  - **Clock algorithm** approximates LRU