Virtual Memory: Demand Paging

COMP 321

Dave Johnson

Copyright © 2026 David B. Johnson

What We Have Done So Far with Paging

- An operating system creates abstractions
- Use hardware support for paging in order to make physical pages appear to be where we want them to appear in the virtual address space
- The operating system kernel creates page tables, and the hardware uses them
- You can use any available physical page for any need
- Eliminates external fragmentation
- Internal fragmentation generally limited to less than page size

Copyright © 2026 David B. Johnson

What is New with Demand Paging?

- Extend this abstraction to make it appear that we have more total memory than we really have
- Uses essentially the same paging hardware we've been using
- The "trick" is that not all pages of "memory" need to be in physical memory at once
- The rest of the "memory" is actually stored on disk (called "backing store")
- Pages are moved from disk into physical memory as needed (***on demand***)
- Physical memory is effectively now just a cache containing a subset of all virtual pages

Copyright © 2026 David B. Johnson

Assumption: The Principle of Locality

Temporal locality

- An executing program is likely to reference ***the same*** memory location again in the near future
- ***Examples***: a variable used in a loop, the instructions within the loop, or a frequently used function

Spatial locality

- An executing program is likely to reference ***nearby*** memory locations in the near future
- ***Examples***: Nearby entries in same array or struct, nearby instructions, other parts of the same page

Copyright © 2026 David B. Johnson

Hardware Requirements: Support in PTE Format

The "valid" bit in each PTE:

V	prot	Page Frame Number (PFN)

- The kernel sets valid = 0 in the PTE if that page is not in physical memory
- Any use of a virtual address within this page by software thus causes an exception
  - The kernel can then read the page from disk into physical memory
  - This type of exception is referred to as a "***page fault***" – but the hardware ***does not*** know what a "page fault" or demand paging is!
- The kernel must also somehow keep track of ***why*** the valid bit was set to 0

Copyright © 2026 David B. Johnson

Basic Handling for a Page Fault

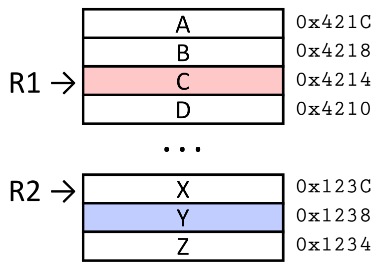
- The ***hardware*** checks PTE valid bit when translating virtual to physical address
- If valid == 0, then the ***hardware*** generates an exception
  - The hardware doesn't know what a "page fault" is!
  - The ***kernel*** figures out that this is a page fault and blocks the process
  - If no free physical page is available, the ***kernel*** selects some virtual page to evict from physical memory to make room (called the "victim" page)
    - The ***kernel*** writes the victim virtual page contents to disk
    - The ***kernel*** clears the valid bit in ***victim page's PTE***
  - The ***kernel*** sets the valid bit and updates pfn in ***needed virtual page's PTE***
  - The ***kernel*** reads the needed virtual page from disk into that physical page
  - The ***kernel*** unblocks the process
- Upon return, ***hardware*** re-executes the instruction that caused the page fault

Copyright © 2026 David B. Johnson

Hardware Requirements: Restartable Instructions

The CPU must have "restartable instructions" (for <u>all</u> instructions):

- VAX example: MOVL (R1)+, -(R2)
  - Moves "longword" (32 bits) from memory pointed to by register R1 (post-increment) to (pre-decrement) memory now pointed to by register R2
  - Equivalent in C to: *--R2 = *R1++

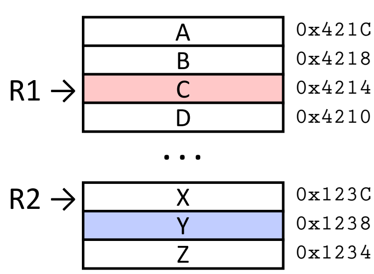


Moves C to Y

Copyright © 2026 David B. Johnson

Hardware Requirements: Restartable Instructions

- What if a page fault occurs when writing to memory where R2 now points?
- The PC register still points to this same MOVL instruction
- Hardware must not increment R1 again/decrement R2 again after page fault
  - Hardware either first undoes side-effects, or remembers which have been done so it does not redo them



Copyright © 2026 David B. Johnson

Other Examples for Restartable Instructions

VAX:

- MOVC3 len, src, dst (similar to memcpy)
- MOVC5 srclen, src, fill, dstlen, dst (also similar to memset)

Intel x86:

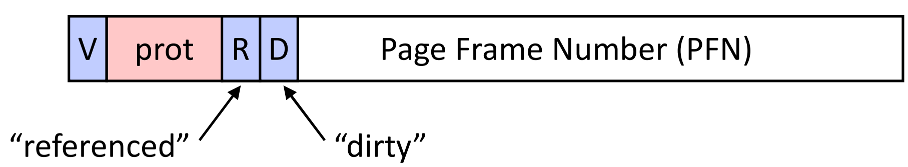
- REP MOVSB
  - src = DS:ESI (x86-64 64-bit: DS:RSI)
  - dst = ES:EDI (x86-64 64-bit: ES:RDI)
  - len = ECX (x86-64 64-bit: RCX)

Without restartable instructions, these instructions might never complete!

9

Other Common, Useful Hardware Support

Two other useful hardware-supported bits in each PTE:



- ***referenced bit***: set in PTE by ***hardware*** when using this PTE for any virtual to physical translation (helps in selecting a "good" victim page)
- ***dirty bit***: set in PTE by ***hardware*** when using this PTE for a reference that will modify contents of page (no need to write victim to disk if not dirty)
- Both bits should be cleared by the ***kernel*** for a new page on a page fault

10

Page Replacement Algorithms

On a page fault, decide the "best" virtual page to evict to make room

- The kernel must bring the needed page into physical memory from disk
- So there must be some available physical page to read it into
- The ***page replacement algorithm*** decides which virtual page will be replaced in physical memory with the needed page
  - The page replacement algorithm ***selects the victim page***
  - The page replacement algorithm ***doesn't*** actually ***do*** page replacement!

How can the kernel decide which page is "best" (or even a "good") choice?

Copyright © 2026 David B. Johnson

Evaluating Page Replacement Algorithms

A ***page reference string*** is a list, in execution order, of the ***virtual page numbers*** referenced during the execution of some program

- Could be recorded from an execution of the program
- Or could be generated by simulating the program's execution

Can be used for comparing one page replacement algorithm to another

- How many page faults occur with this page reference string when using page replacement Algorithm A vs. Algorithm B?
- We can compare many algorithms on many different page reference strings
- ***Only the virtual page numbers matter, not the full virtual addresses***

Copyright © 2026 David B. Johnson

Page Replacement: Random

Select some virtual page in physical memory at random as the victim

- Easy to implement
- No extra information (e.g., history) to keep track of
- Freedom from ***always*** making a "bad" (or the "worst") choice

But what about the principle of locality?

- Principle of locality is a good description of typical real program behavior
- Temporal locality and spatial locality
- We can often make better replacement decisions by exploiting the assumption of locality

Page Replacement: Optimal (MIN or OPT)

Select the virtual page in physical memory whose next reference is the farthest into the future (in the page reference string)

- ***Not implementable, but a good point of comparison***

Example with page reference string

(2) (1) (4) 2 (3) (0) 4 2 (5) 4

with 4 pages of physical memory, ***with all 4 pages starting empty***

PFN	0	1	2	3
Current VPN	2	1	4	3
	5			0

Total page faults = 4 + 2 = 6 ***No real algorithm can do better!***

Page Replacement: First-in-First-out (FIFO)

Select the virtual page in physical memory that was <u>brought into physical memory longest ago</u>

Example with the same page reference string:

(2) (1) (4) 2 (3) (0) 4 (2) (5) (4)

PFN	**0**	**1**	**2**	**3**
Current VPN	2	1	4	3
	0	2	5	4

Total page faults = 4 + 4 = 8 (2 more than when using MIN)

Copyright © 2026 David B. Johnson

Page Replacement: Least-Recently-Used (LRU)

Select the virtual page in physical memory that was <u>last referenced longest ago</u>

Example with the same page reference string:

(2) (1) (4) 2 (3) (0) 4 2 (5) 4

PFN	**0**	**1**	**2**	**3**
Current VPN	2	1	4	3
		0		5

Total page faults = 4 + 2 = 6 (equal to when using MIN)

Copyright © 2026 David B. Johnson

FIFO and LRU Implementation Issues

Implementing FIFO replacement is easy

- The kernel is involved in bringing each new virtual page into physical memory
- The kernel can keep track of the relative ordering of these events since it can see that ordering

But implementing LRU requires some kind of hardware support

- The kernel is only involved in (and thus can only see) references that resulted in page faults
- Only the hardware is involved in each other reference, even though those references affect the LRU ordering

Copyright © 2026 David B. Johnson

LRU Hardware Implementation using Counters

Page reference string

2	1	4	2	3	0	4	2	5	4
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)

PFN	0	1	2	3
Current VPN	2 (7)	1 (1)	4 (9)	3 (4)
		0 (5)		5 (8)

The hardware must maintain the counter and have a place to save the counter value associated with ***every*** single physical memory page!

- ***Too much space and time overhead***

Copyright © 2026 David B. Johnson

LRU Hardware Implementation using Counters

Requires hardware support

- Hardware must increment the counter on each memory reference
- Hardware must store the current counter value for each physical memory page, updated when that page is referenced
- Hardware must be able to tell the kernel, or the kernel must be able to find, the minimum of these values to be the victim page
- And where are those counter values stored?
  – Not compatible with the normal way that physical memory is packaged, sold, and installed
  – And must be ***fast*** memory for storing the counter values

Copyright © 2026 David B. Johnson

LRU Hardware Implementation using a Stack

Page reference string

2 1 4 2 3 0 4 2 5 4

Stack	2	1	4	2	3	0	4	2	5	4
		2	1	4	2	3	0	4	2	5
			2	1	4	2	3	0	4	2
					1	4	2	3	0	0
						1			3	

The hardware must store this stack somewhere and rearrange it with ***every*** single memory reference!

- ***Too much space and time overhead***

Copyright © 2026 David B. Johnson

LRU Hardware Implementation using a Stack

Requires hardware support

- Hardware must maintain (either push or reorder) the stack entries on each memory reference
- High overhead (particularly for large number of physical memory pages) on every memory reference
- Hardware must be able to tell the kernel, or the kernel must be able to find, the value forced off the bottom, to be the victim page
- And where is this stack stored?
  - Not compatible with the normal way that physical memory is packaged, sold, and installed
  - And must be ***fast*** memory for storing the stack

Copyright © 2026 David B. Johnson

Many Replacement Algorithms "Approximate" LRU

LRU is often the "best" basis for a page replacement algorithm

- LRU "approximates" predicting the future
- The principle of locality: temporal local (and spatial locality)
- Past behavior is a "good" predictor of future performance

But some problems

- LRU is not implementable in software (in any practical way)
- And so requires hardware support (that itself is not practical)

So use an ***approximation*** of an ***approximation*** of predicting the future!

- Many algorithms, and many variants of them

Copyright © 2026 David B. Johnson

Approximation: Reference-Bit-History Algorithm

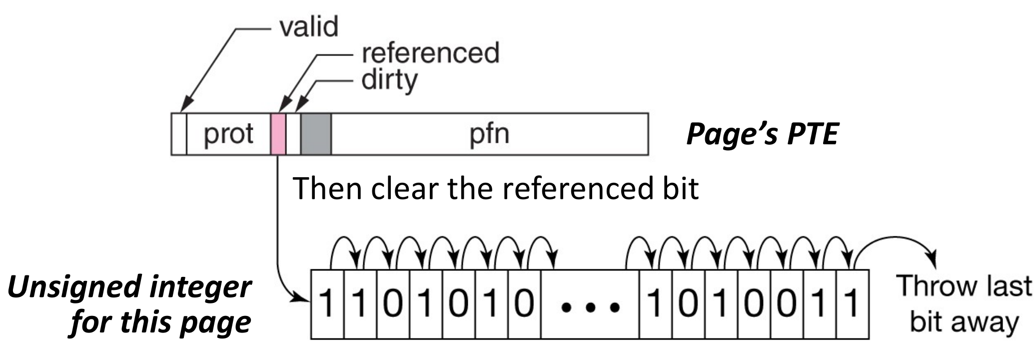
At boot time

- Create an array of *P* ***unsigned integers***, where *P* is the total number of pages of ***physical*** memory, and initialize each integer to 0
- The number of bits in each of the integers is a tradeoff

Periodically (e.g., every k clock interrupts)

- For every page of ***physical*** memory that is in use as some ***virtual*** page:
  - ***Shift*** the corresponding unsigned integer ***right*** by 1 bit
  - ***Find*** the PTE for that ***virtual*** page
  - ***Copy*** value of that PTE's referenced bit to ***high order bit*** of that integer
  - ***Clear*** that PTE's referenced bit

Reference-Bit-History Algorithm



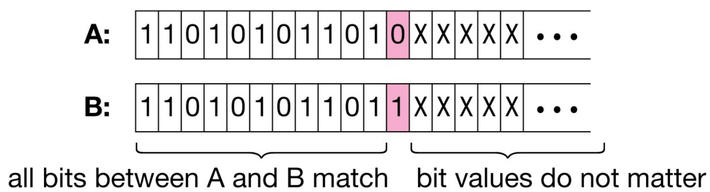
Integer thus contains a history of sampled value of that page's referenced bit

This scheme is also known as "***Aging***" or "***Additional-Reference-Bits***"

Reference-Bit-History Algorithm

Selecting the victim page

- The physical memory page corresponding to the ***global minimum unsigned integer*** (one value for each page) is selected as the victim page
- Example: **A** < **B** if and only if:



- This means the recent history of page **A** and page **B** are the same, but in the next-most recent sample time interval, **B** was referenced but **A** was not

Copyright © 2026 David B. Johnson

Reference-Bit-History Algorithm

This algorithm actually results in <u>perfect</u> LRU, <u>except for</u>

- It only keeps a ***limited number of bits*** of history
  - If all bits of the unsigned integer for A and for B are the same, you can't tell which is LRU since you can't compare at times earlier than that
  - But the principle of locality only refers to "in the near future" (and thus corresponding to in the recent past)
- It only samples the reference bits ***at periodic intervals***
  - If some bit (a reference bit sample) for A and for B are both 1, you can't order the references ***within*** that sampling interval
  - But the principle of locality only refers to "in the near future," not to an exact ordering at an infinitely small time scale

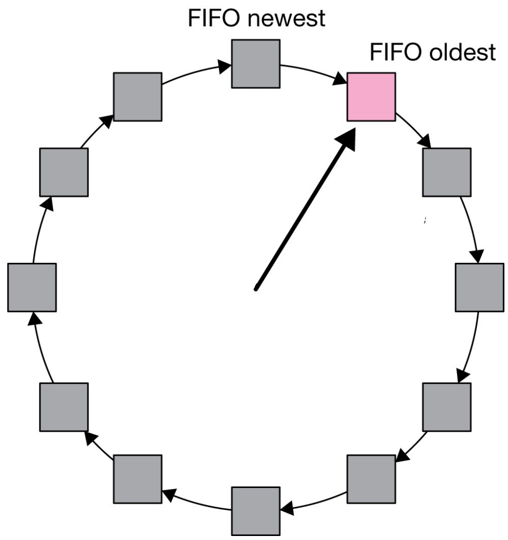
Copyright © 2026 David B. Johnson

Approximation: Second-Chance / CLOCK Algorithm

A modified form of FIFO to make it behave more "LRU-like"

Imagine all ***physical memory pages*** arranged in a circular list in ***FIFO*** order

- Like around the face of a clock
- The "hand" of the clock points to the FIFO oldest page



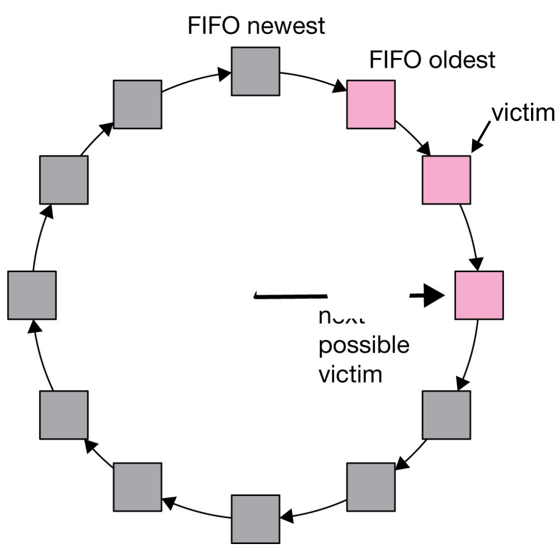
Copyright © 2026 David B. Johnson

Second-Chance / CLOCK Algorithm

Victim selection

```
victim = -1;
while (victim == -1) {
        if (PTE[hand].ref == 1)
                PTE[hand].ref = 0;
        else
                victim = hand;
        advance hand;
}
```

If no victim is found in the first full revolution, then the second revolution is effectively pure FIFO



Copyright © 2026 David B. Johnson

“Enhanced” Second-Chance

Prefer a victim page for which the PTE dirty bit is not set

- Avoids the expense of writing page to disk, so faster at least for ***<u>this</u>*** page fault
- On the first revolution of the hand

Referenced	Dirty	
0	0	Victim page found, stop looking
0	1	Do not change bits, advance the hand
1	0	Set referenced = 0, advance the hand
1	1	Set referenced = 0, advance the hand

- After the first revolution of the hand, all referenced bits are now 0
  – During the second revolution, victim is the first page for which dirty is 0
- On the third revolution, victim is the first page (thus pure FIFO)

Copyright © 2026 David B. Johnson