CS111, Lecture 24
The Clock Algorithm

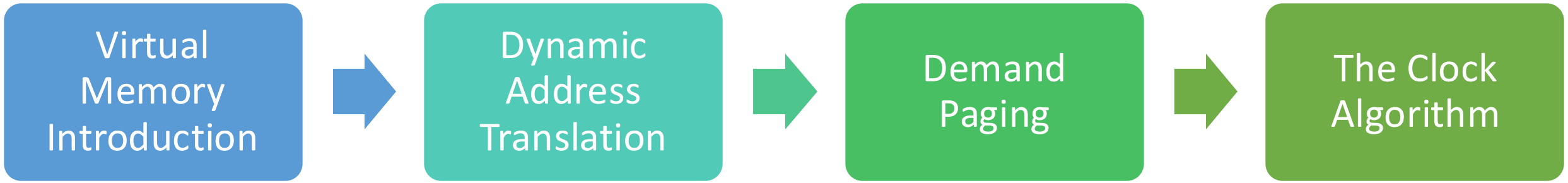
Optional reading:

Operating Systems: Principles and Practice (2nd Edition): Chapter 9

Today: if we need to kick out a page, which one do we pick?

CS111 Topic 4: Virtual Memory

Virtual Memory *- How can one set of memory be shared among several processes? How can the operating system manage access to a limited amount of system memory?*



assign6: implement *demand paging* system to translate addresses and load/store memory contents for programs as needed.

Learning Goals

- Learn about tradeoffs in approaches for choosing pages to kick out of memory
- Walk through the implementation of the clock algorithm, one algorithm for choosing which page to throw out

Plan For Today

- **Recap:** Demand Paging
- The Clock Algorithm
- What about when the OS Runs?
- Virtual Memory summary

Plan For Today

- **Recap: Demand Paging**
- The Clock Algorithm
- What about when the OS runs?
- Virtual Memory summary

Demand Paging

If memory is in high demand, we could fill up all of memory, since a process needs all its pages in memory to run. What should we do in that case?

- Prohibit further program memory requests? Not ideal.
- Another idea – what if we kicked out a page and used that page? We could save a page to disk, use the page for new data, and load the old data back in to a physical page later if it's still needed.

Overall goal: make physical memory look larger than it is.

Demand Paging

If we need another page but memory is full:

1. Pick a page to kick out
2. Write it to disk
3. Mark the old page map entry as not present
4. Update the new page map entry to be present and map to this physical page

Demand Paging

If the program accesses a page that was swapped to disk:

1. Triggers a page fault (not-present page accessed)
2. We see disk swap contains data for this page
3. Get a new physical page (perhaps kicking out another one)
4. Load the data from disk into that page
5. Update the page map with this new mapping

Thrashing

Demand paging can provide big benefits – but we can encounter *thrashing*; when the pages being actively used don't all fit in memory, and the system will spend all its time reading and writing pages to/from disk and won't get much work done.

- The page we kick to disk will be needed very soon, so we will bring it back and kick another page, which will be needed very soon, etc….
- Progress of the program will make it look like access time of memory is as slow as disk, rather than disks being as fast as memory. ☹
- With personal computers, users can notice thrashing and kill some processes

Page Fetching

Now we have a mechanism to allow programs to run without all their information in memory. But even if there is space, when should we bring pages into memory?

- Most modern OSes start with no pages loaded, load pages when referenced ("demand fetching").
- Alternative: *prefetching* - try to predict when pages will be needed and load them ahead of time (requires predicting the future…)

Demand Paging Behaviors

Two additional details about demand paging:

- We don't *always* need to write a swapped-out page to disk (e.g., read-only code pages can always be loaded from executable)
- A page may have initial data even if it's never been accessed before (e.g., initialized global variables at program start.)

The pages for a process divide into three groups:

1. **Read-only code pages**: program code
2. **Initialized data pages:** program data with initial values (e.g., globals)
3. **Uninitialized data pages:** e.g., stack, heap

Demand Paging Behaviors

Detail #1: we don't *always* need to write a swapped-out page to disk (e.g., read-only code pages can always be loaded from executable).

1. **Read-only code pages:** program code, doesn't change – no need to store in swap, can re-read from executable
2. **Initialized data pages:** program data with initial values (e.g., globals) – save to swap since contents may have changed
3. **Uninitialized data pages:** e.g., stack, heap – save to swap as needed

Demand Paging Behaviors

Detail #2: a page may have initial data even if it's never been accessed before (e.g., initialized global variables at program start.)

1. **Read-only code pages:** program code – on first access, expected to have data
2. **Initialized data pages:** program data with initial values (e.g., globals) – on first access, expected to have data
3. **Uninitialized data pages:** e.g., stack, heap – no set initial contents, on first access, clear to all 0s

Plan For Today

- **Recap:** Demand Paging
- **The Clock Algorithm**
- What about when the OS runs?
- Virtual Memory summary

Page Replacement

If we need another physical page but all memory is used, which page should we throw out? How do we pick?

- Random? (works surprisingly well!)
- FIFO? (throw out page that's been in memory the longest) – fairness
- Would be nice if we could pick page whose next access is farthest in the future, but we'd need to predict the future…
- LRU (least-recently-used)? Replace page that was accessed the longest time ago.

Page Replacement

If we need another physical page but all memory is used, which page should we throw out? How do we pick?

- Random? (works surprisingly well!)
- FIFO? (throw out page that's been in memory the longest) – fairness
- Would be nice if we could pick page whose next access is farthest in the future, but we'd need to predict the future…
- **LRU (least-recently-used)? Replace page that was accessed the longest time ago.**

Least-Recently-Used

How could we know which page was the least-recently used?

- Store clock time for each page on each reference?
- Scan all pages to find oldest one?

Alternative: just find an old page, not necessarily the oldest.

The **clock algorithm** is one implementation of this idea.

Clock algorithm key idea: rotate through pages until we find one that hasn't been referenced since the *last time* we checked it. ("second chance algorithm")

Clock Algorithm

New **reference bit** tracks whether a page has been referenced recently.

- Set to 1 whenever that page is read or written
- Set to 0 by clock algorithm if it considered kicking it out, but will instead circle back again next time

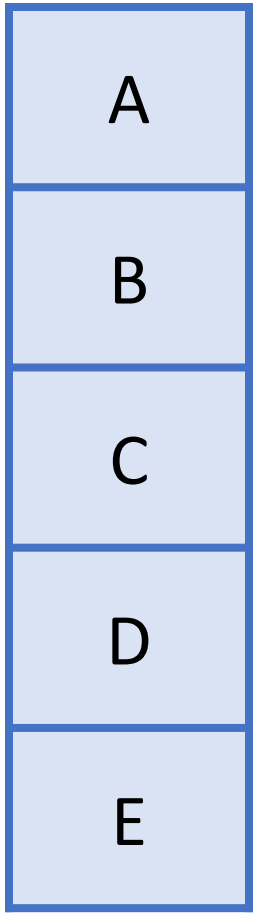
The clock algorithm cycles through pages (looping back around over time – may be run multiple times) until it chooses a page to kick out.

"reference" bit

	Physical page #	WR?	PR?	R
7	E	1	1	0
6	D	1	1	1
5	X	X	0	X
4	X	X	0	X
3	X	X	0	X
2	C	1	1	0
1	B	0	1	1
0	A	0	1	1

Page Map

Clock Algorithm



Physical Pages

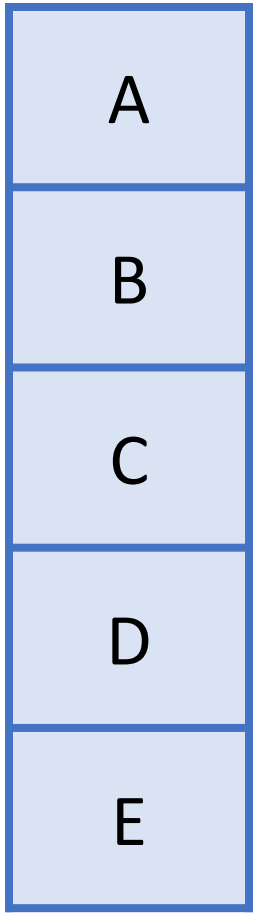
Let's say the system looks as follows, and a program requests mapping page 5, but we have no more physical pages. This triggers the clock algorithm.

"reference" bit

	Physical page #	WR?	PR?	R
7	E	1	1	0
6	D	1	1	1
5	X	X	0	X
4	X	X	0	X
3	X	X	0	X
2	C	1	1	0
1	B	0	1	1
0	A	0	1	1

Page Map

Clock Algorithm



Physical Pages

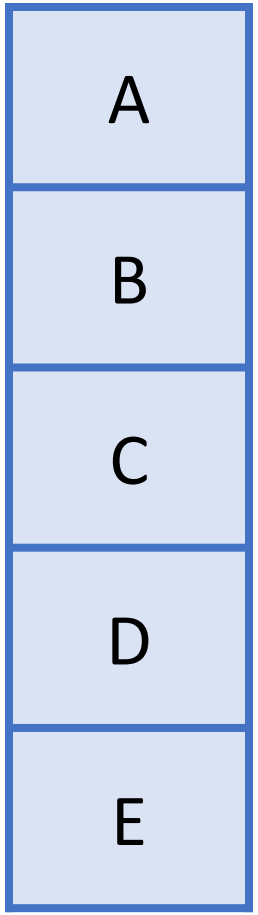
Clock algorithm starts where it left off the previous time it was run (or just at the beginning the first time)

"reference" bit

	Physical page #	WR?	PR?	R
7	E	1	1	0
6	D	1	1	1
5	X	X	0	X
4	X	X	0	X
3	X	X	0	X
2	C	1	1	0
1	B	0	1	1
0	A	0	1	1

Page Map

Clock Algorithm



Physical Pages

Was this page accessed recently (reference = 1)? If so, set reference = 0 and continue.

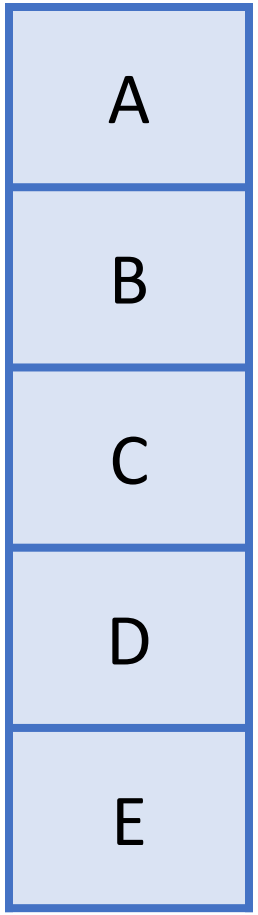
"We'll leave this page for now – but if we come back and it's still unused, we'll kick it out."

"reference" bit

	Physical page #	WR?	PR?	R
7	E	1	1	0
6	D	1	1	1
5	X	X	0	X
4	X	X	0	X
3	X	X	0	X
2	C	1	1	0
1	B	0	1	1
0	A	0	1	1

Page Map

Clock Algorithm



Physical Pages

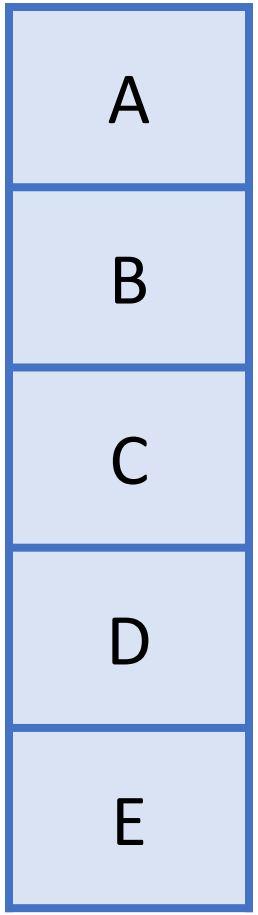
Was this page accessed recently (reference = 1)? If so, set reference = 0 and continue.

"reference" bit

	Physical page #	WR?	PR?	R
7	E	1	1	0
6	D	1	1	1
5	X	X	0	X
4	X	X	0	X
3	X	X	0	X
2	C	1	1	0
1	B	0	1	1
0	A	0	1	1

Page Map

Clock Algorithm



Physical Pages

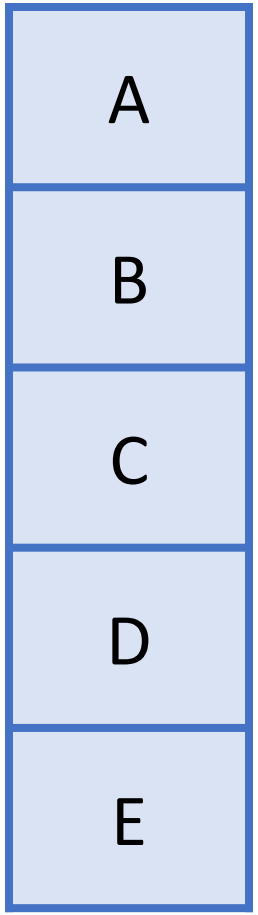
Was this page accessed recently (reference = 1)? If so, set reference = 0 and continue.

"reference" bit

	Physical page #	WR?	PR?	R
7	E	1	1	0
6	D	1	1	1
5	X	X	0	X
4	X	X	0	X
3	X	X	0	X
2	C	1	1	0
1	B	0	1	1
0	A	0	1	0

Page Map

Clock Algorithm



Physical Pages

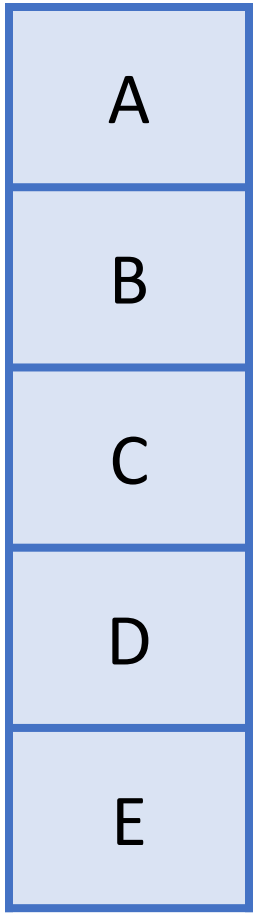
Was this page accessed recently (reference = 1)? If so, set reference = 0 and continue.

"reference" bit

	Physical page #	WR?	PR?	R
7	E	1	1	0
6	D	1	1	1
5	X	X	0	X
4	X	X	0	X
3	X	X	0	X
2	C	1	1	0
1	B	0	1	1
0	A	0	1	0

Page Map

Clock Algorithm



Physical Pages

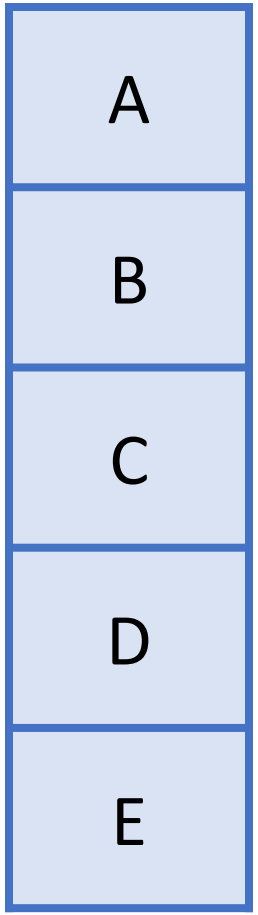
Was this page accessed recently (reference = 1)? If so, set reference = 0 and continue.

"reference" bit

	Physical page #	WR?	PR?	R
7	E	1	1	0
6	D	1	1	1
5	X	X	0	X
4	X	X	0	X
3	X	X	0	X
2	C	1	1	0
1	B	0	1	0
0	A	0	1	0

Page Map

Clock Algorithm



Physical Pages

Was this page accessed recently (reference = 1)? If not, this is the one we should remove.

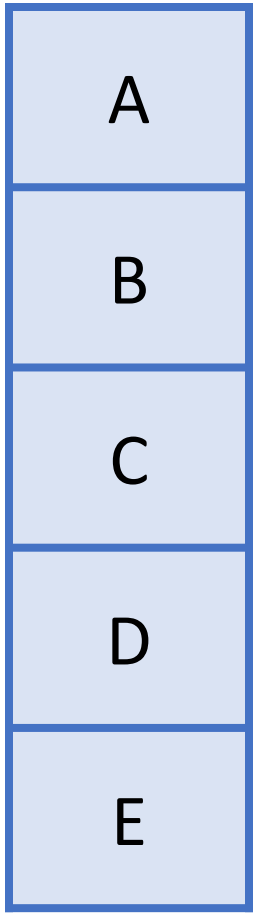
"This page hasn't been used 'recently' - let's remove it.

"reference" bit

	Physical page #	WR?	PR?	R
7	E	1	1	0
6	D	1	1	1
5	X	X	0	X
4	X	X	0	X
3	X	X	0	X
2	C	1	1	0
1	B	0	1	0
0	A	0	1	0

Page Map

Clock Algorithm



Physical Pages

Was this page accessed recently (reference = 1)? If not, this is the one we should remove.

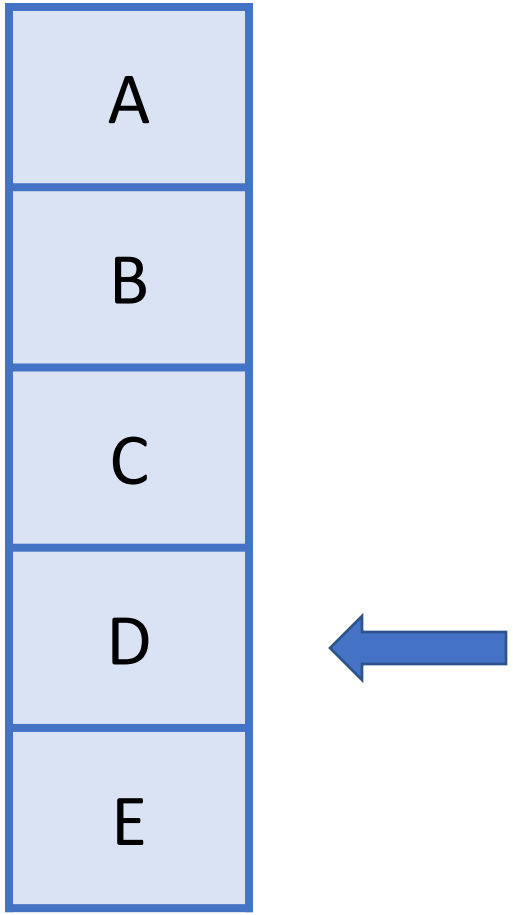
"This page hasn't been used 'recently' - let's remove it.

"reference" bit

	Physical page #	WR?	PR?	R
7	E	1	1	0
6	D	1	1	1
5	C	1	1	1
4	X	X	0	X
3	X	X	0	X
2	X	X	0	X
1	B	0	1	0
0	A	0	1	0

Page Map

Clock Algorithm



Physical Pages

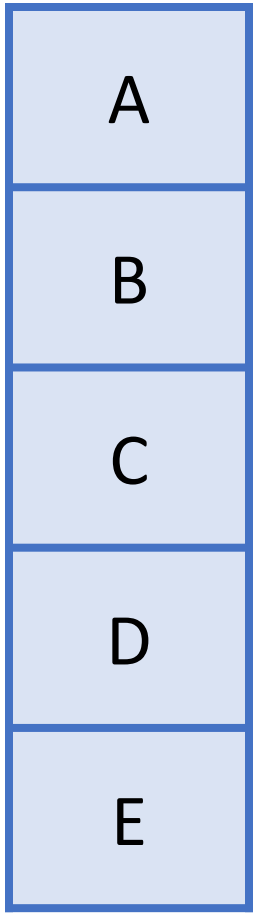
Now the clock algorithm stops, and **we remember the position of the hand for next time it runs.**

"reference" bit

	Physical page #	WR?	PR?	R
7	E	1	1	0
6	D	1	1	1
5	C	1	1	1
4	X	X	0	X
3	X	X	0	X
2	X	X	0	X
1	B	0	1	0
0	A	0	1	0

Page Map

Clock Algorithm



Physical Pages

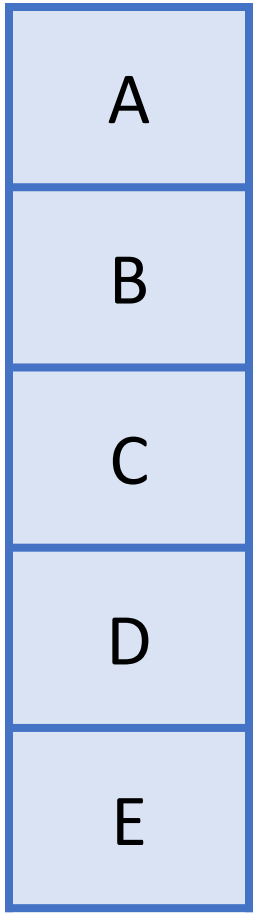
In the meantime, the program resumes running, and a long time could pass between runs of the clock algorithm. During that time, pages could be accessed, meaning reference bits may change.

"reference" bit

	Physical page #	WR?	PR?	R
7	E	1	1	0
6	D	1	1	1
5	C	1	1	1
4	X	X	0	X
3	X	X	0	X
2	X	X	0	X
1	B	0	1	0
0	A	0	1	0

Page Map

Clock Algorithm



Physical Pages

In the meantime, the program resumes running, and a long time could pass between runs of the clock algorithm. During that time, pages could be accessed, meaning reference bits may change.

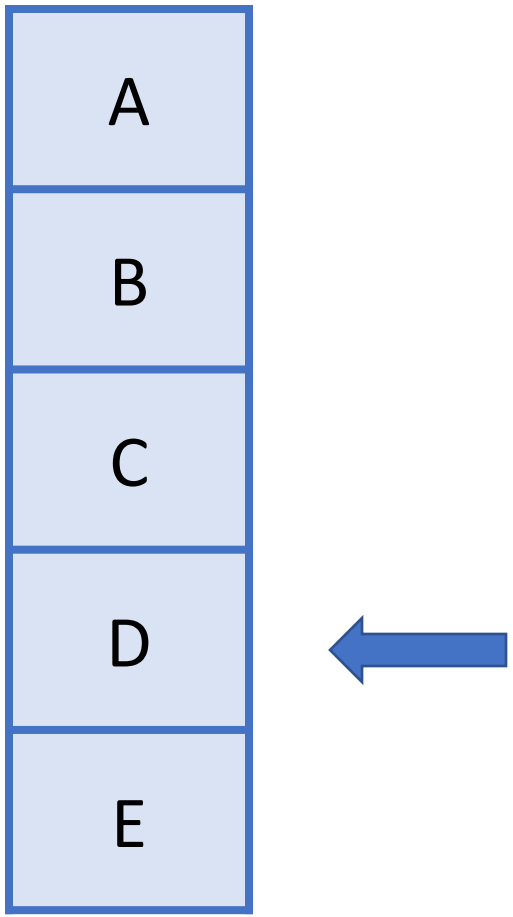
"reference" bit

	Physical page #	WR?	PR?	R
7	E	1	1	1
6	D	1	1	1
5	C	1	1	1
4	X	X	0	X
3	X	X	0	X
2	X	X	0	X
1	B	0	1	1
0	A	0	1	0

Page Map

Clock Algorithm

Let's say the program now requests mapping page 4.



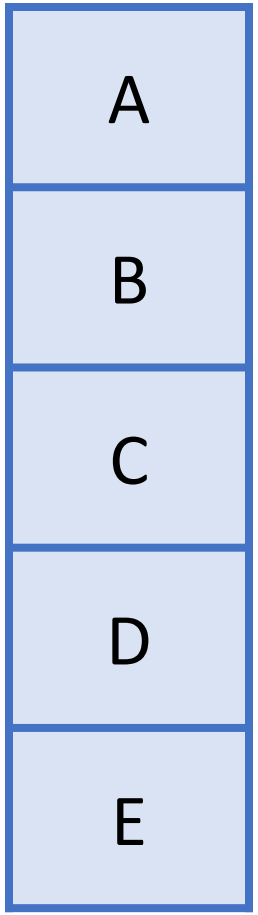
Physical Pages

"reference" bit

	Physical page #	WR?	PR?	R
7	E	1	1	1
6	D	1	1	1
5	C	1	1	1
4	X	X	0	X
3	X	X	0	X
2	X	X	0	X
1	B	0	1	1
0	A	0	1	0

Page Map

Clock Algorithm



Physical Pages

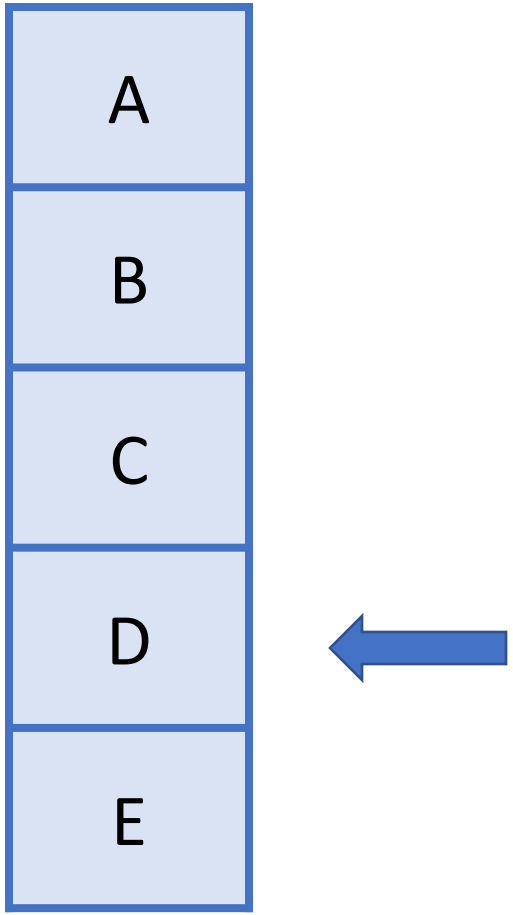
Was this page accessed recently (reference = 1)? If so, set reference = 0 and continue.

"reference" bit

	Physical page #	WR?	PR?	R
7	E	1	1	1
6	D	1	1	1
5	C	1	1	1
4	X	X	0	X
3	X	X	0	X
2	X	X	0	X
1	B	0	1	1
0	A	0	1	0

Page Map

Clock Algorithm



Physical Pages

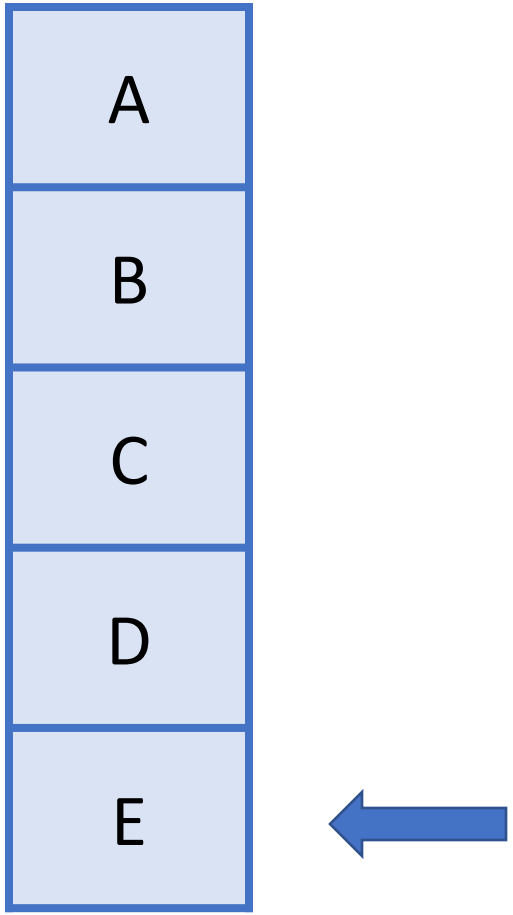
Was this page accessed recently (reference = 1)? If so, set reference = 0 and continue.

"reference" bit

	Physical page #	WR?	PR?	R
7	E	1	1	1
6	D	1	1	0
5	C	1	1	1
4	X	X	0	X
3	X	X	0	X
2	X	X	0	X
1	B	0	1	1
0	A	0	1	0

Page Map

Clock Algorithm



Physical Pages

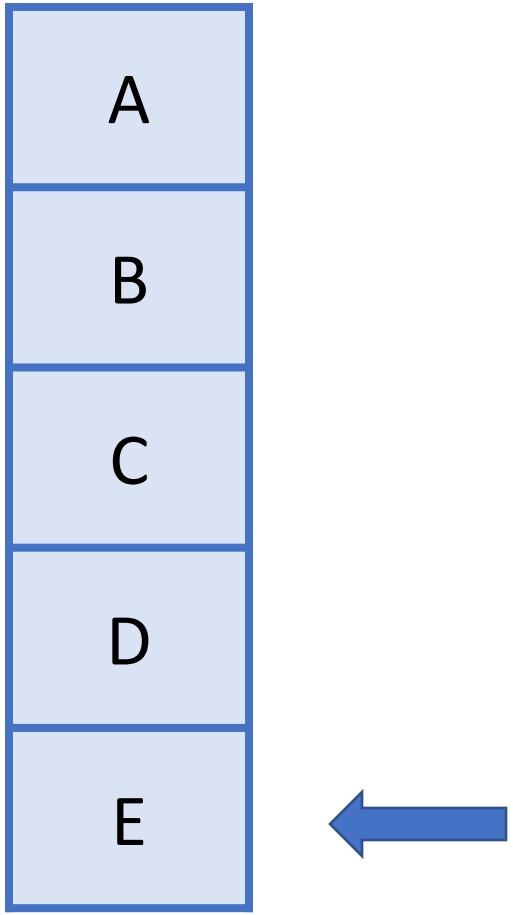
Was this page accessed recently (reference = 1)? If so, set reference = 0 and continue.

"reference" bit

	Physical page #	WR?	PR?	R
7	E	1	1	1
6	D	1	1	0
5	C	1	1	1
4	X	X	0	X
3	X	X	0	X
2	X	X	0	X
1	B	0	1	1
0	A	0	1	0

Page Map

Clock Algorithm



Physical Pages

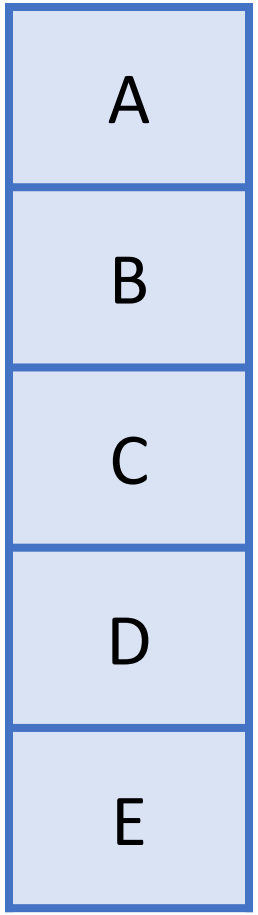
Was this page accessed recently (reference = 1)? If so, set reference = 0 and continue.

"reference" bit

	Physical page #	WR?	PR?	R
7	E	1	1	0
6	D	1	1	0
5	C	1	1	1
4	X	X	0	X
3	X	X	0	X
2	X	X	0	X
1	B	0	1	1
0	A	0	1	0

Page Map

Clock Algorithm



Physical Pages

Was this page accessed recently (reference = 1)? If not, this is the one we should remove.

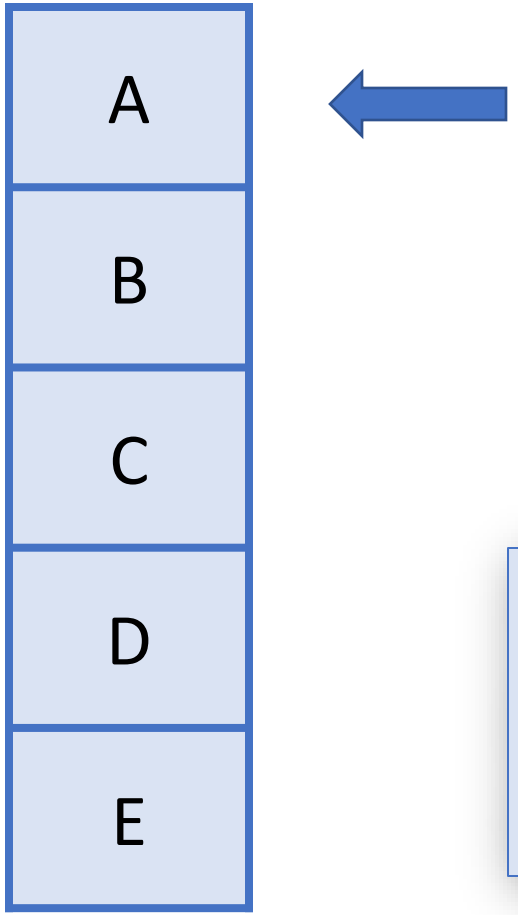
"This page hasn't been used 'recently' - let's remove it.

"reference" bit

	Physical page #	WR?	PR?	R
7	E	1	1	0
6	D	1	1	0
5	C	1	1	1
4	X	X	0	X
3	X	X	0	X
2	X	X	0	X
1	B	0	1	1
0	A	0	1	0

Page Map

Clock Algorithm



Physical Pages

Was this page accessed recently (reference = 1)? If not, this is the one we should remove.

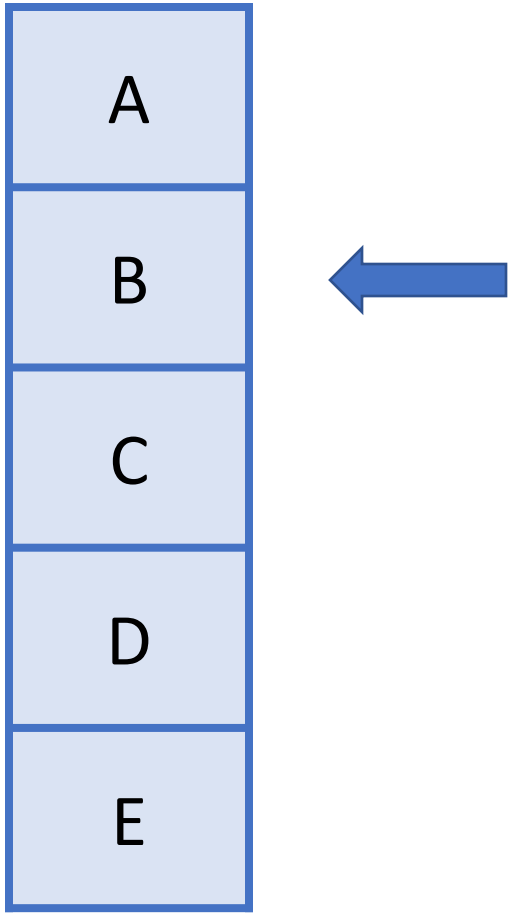
"This page hasn't been used 'recently' - let's remove it.

"reference" bit

	Physical page #	WR?	PR?	R
7	E	1	1	0
6	D	1	1	0
5	C	1	1	1
4	A	1	1	1
3	X	X	0	X
2	X	X	0	X
1	B	0	1	1
0	X	X	0	X

Page Map

Clock Algorithm



Physical Pages

Now the clock algorithm stops, and **we remember the position of the hand for next time it runs.**

"reference" bit

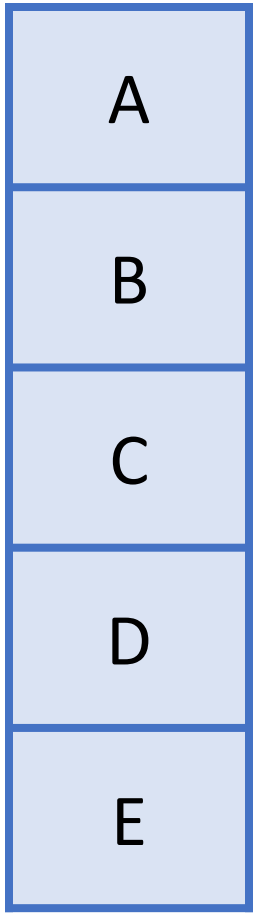
	Physical page #	WR?	PR?	R
7	E	1	1	0
6	D	1	1	0
5	C	1	1	1
4	A	1	1	1
3	X	X	0	X
2	X	X	0	X
1	B	0	1	1
0	X	X	0	X

Page Map

Clock Algorithm Summary

- We add a *reference* bit: set whenever a page is read or written
  - Set to 1 whenever that page is read or written
  - Set to 0 if clock algorithm considers kicking it out, but instead circles back later
- When physical memory is full and we need to choose a page to remove, run the clock algorithm.
- Clock hand "sweeps" over pages, rotating back to start if reaching the end.
- Every time the hand visits a page, we ask: "Has this page been referenced since the last time the clock hand swept over it?"
  - **If YES (reference = 1):** mark it as not referenced, and advance clock hand ("sweep")
  - **If NO (reference = 0):** choose it for removal, advance clock hand, stop clock algorithm
- The clock hand position is saved for the next time the algorithm runs
- "Second chance" algorithm – reference bit = "# of free passes left"
- Needs mechanism to map from physical pages back to virtual pages

Clock Algorithm



Physical Pages

Let's say we have a new setup, and the clock hand starts at C. Which page will the clock algorithm choose to reuse when run next?

Respond on PollEv:
pollev.com/cs111 or text CS111 to 22333 once to join.

"reference" bit

	Physical page #	WR?	PR?	R
7	E	1	1	1
6	D	1	1	1
5	C	1	1	1
4	A	1	1	1
3	X	X	0	X
2	X	X	0	X
1	B	0	1	1
0	X	X	0	X

Page Map

Which physical page will be reused next?

Page A 0%

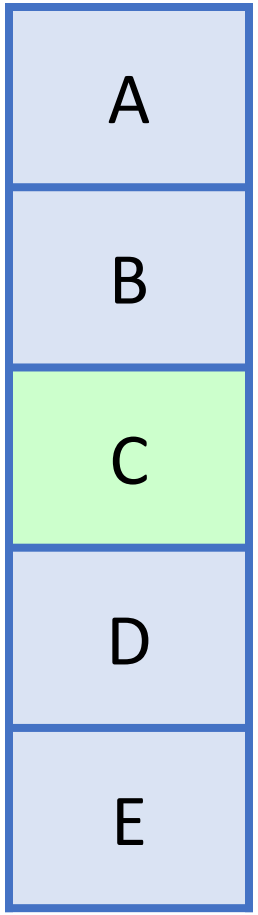
Page B 0%

Page C 0%

Page D 0%

Page E 0%

Clock Algorithm



Physical Pages

Let's say we have a new setup, and the clock hand starts at C. Which page will the clock algorithm choose to reuse when run next?

"reference" bit

	Physical page #	WR?	PR?	R
7	E	1	1	1
6	D	1	1	1
5	C	1	1	1
4	A	1	1	1
3	X	X	0	X
2	X	X	0	X
1	B	0	1	1
0	X	X	0	X

Page Map

Page Replacement

How does page replacement work if there are multiple processes running?

- *Per-process replacement:* each process has separate pool of physical pages, and a page fault in a process can only replace one of its own pages. But how many physical pages should each process get?
- *Global replacement* (most common): all pages from all processes in single replacement pool. A page fault in one process can kick out a page in another process.

Plan For Today

- **Recap:** Demand Paging
- The Clock Algorithm
- **What about when the OS runs?**
- Virtual Memory summary

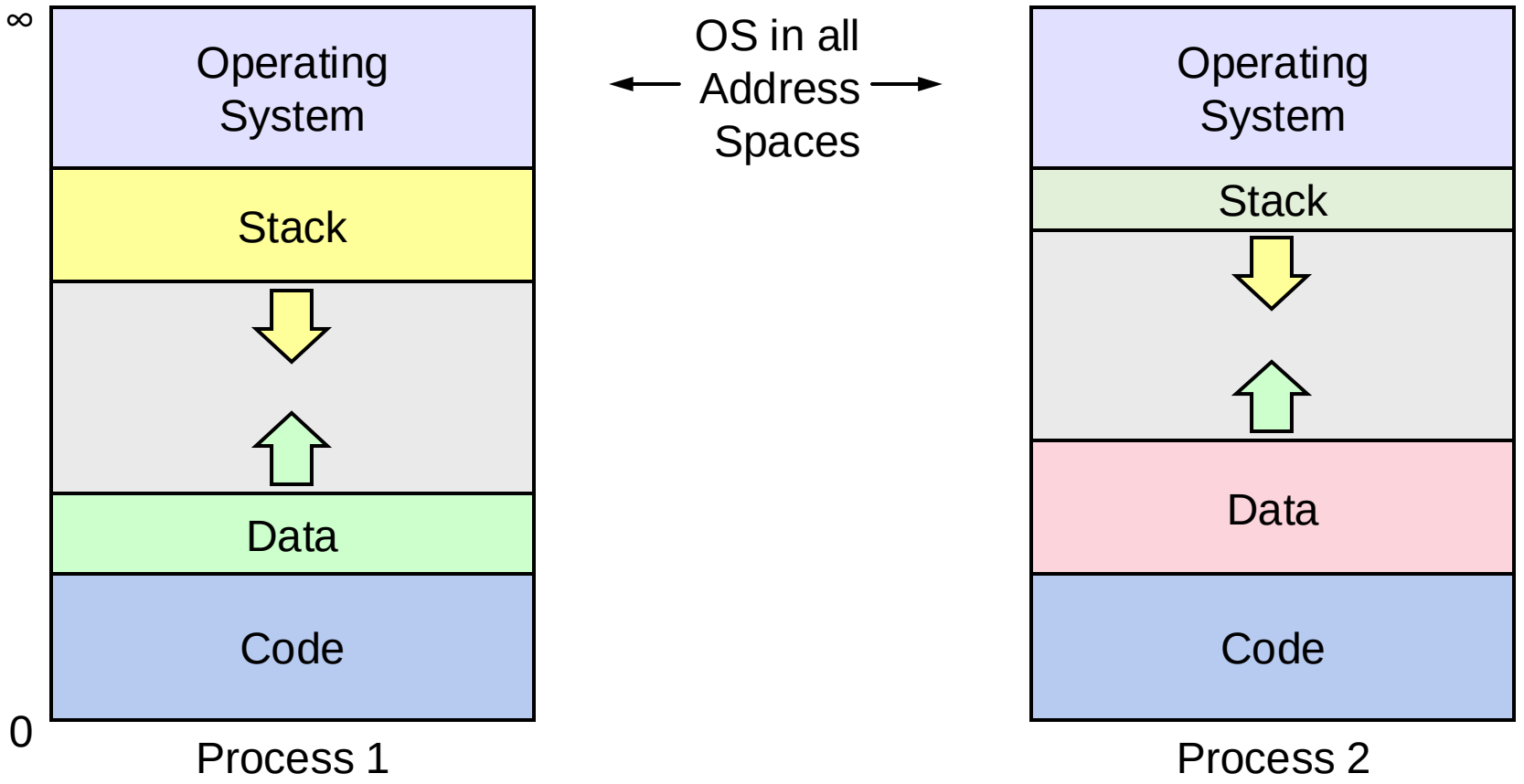
OS Execution

How does virtual memory work when the OS runs?

Challenge: whenever a process makes a system call, any address parameters are *virtual* addresses, and the OS may need to access data from the process's virtual address space.

- One option: the OS runs "unmapped" - it works directly with physical memory (no virtual->physical mappings). But then it must do any virtual address translations itself.
- Another (most common in modern systems) option: the OS runs mapped in *every* process's virtual address space. Then it piggybacks on automatic MMU translation.

OS and User in Same Address Space



OS Execution

How does virtual memory work when the OS runs?

OS has space in every process's virtual address space. Not a duplicate of OS; every virtual space could map to same physical memory.

Problem: don't want user program accessing OS pages.

Solution: new bit in page table that marks kernel-only pages. When in user mode, not accessible, but accessible when OS is running.

Plan For Today

- **Recap:** Demand Paging
- More Demand Paging Details
- The Clock Algorithm
- What about when the OS runs?
- **Virtual Memory summary**

Virtual Memory

- Virtual memory is an example of “OS magic” – very powerful mechanism
- Virtualization: making one thing look like another – separation between appearance and reality
- OS can manage physical memory how it wants (e.g. swap to disk), invisible to user programs

Goals:

- **Multitasking** – allow multiple processes to be memory-resident at once
- **Transparency** – no process should need to know memory is shared. Each must run regardless of the number and/or locations of processes in memory.
- **Isolation** – processes must not be able to corrupt each other
- **Efficiency** (both of CPU and memory) – shouldn’t be degraded badly by sharing

CS111 Topic 4: Virtual Memory

Virtual Memory - *How can one set of memory be shared among several processes? How can the operating system manage access to a limited amount of system memory?*

Why is answering this question important?

- We can understand one of the most "magical" responsibilities of OSes – making one set of memory appear as several!
- Exposes challenges of allowing multiple processes to share memory while remaining isolated
- Allows us to understand exactly what happens when a program accesses a memory address

assign6: implement *paging/demand paging* system to translate addresses and load/store memory contents for programs as needed.

Recap

- Recap: Demand Paging
- The Clock Algorithm
- What about when the OS runs?
- Virtual Memory summary

Lecture 24 takeaway: There are many different policies to choose a page to kick out when memory is full. The clock algorithm is one approximation of LRU to pick an old page to remove.