UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Introduction to Operating Systems

Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

Page Replacement in Real Systems

Questions answered in this lecture:

How can the LRU page be approximated efficiently?

How can users discover the page replacement algorithm of the OS?

What page replacement algorithms are used in existing systems?

Implementing LRU

Software Perfect LRU

- OS maintains ordered list of physical pages by reference time
- When page is referenced: Move page to front of list
- When need victim: Pick page at back of list
- Trade-off: Slow on memory reference, fast on replacement

Hardware Perfect LRU

- Associate register with each page
- When page is referenced: Store system clock in register
- When need victim: Scan through registers to find oldest clock
- Trade-off: Fast on memory reference, slow on replacement (especially as size of memory grows)

In practice, do not implement Perfect LRU

- LRU is an approximation anyway, so approximate more
- Goal: Find an old page, but not necessarily the very oldest

Clock (Second Chance) Algorithm

Hardware

- Keep `use` (or `reference`) bit for each page frame
- When page is referenced: set `use` bit

Operating System

- Page replacement: Look for page with `use` bit cleared (has not been referenced for awhile)
- Implementation:
  - Treat pages as circular buffer
  - Keep pointer to last examined page frame
  - Traverse pages in circular buffer
  - Clear `use` bits as search
  - Stop when find page with cleared `use` bit, replace this page

Clock Algorithm Example

What if clock hand is sweeping very fast?
What if clock hand is sweeping very slow?

Clock Extensions

Replace multiple pages at once

- Intuition: Expensive to run replacement algorithm and to write single block to disk
- Find multiple victims each time

Two-handed clock

- Intuition:
  - If takes long time for clock hand to sweep through pages, then all use bits might be set
  - Traditional clock cannot differentiate between usage of different pages
- Allow smaller time between clearing use bit and testing
  - First hand: Clears use bit
  - Second hand: Looks for victim page with use bit still cleared

More Clock Extensions

Add software counter ("chance")

- Intuition: Better ability to differentiate across pages (how much they are being accessed)
- Increment software counter `chance` if `use` bit is 0
- Replace when `chance` exceeds some specified limit

Use `dirty` bit to give preference to dirty pages

- Intuition: More expensive to replace dirty pages
  - Dirty pages must be written to disk, clean pages do not
- Replace pages that have `use` bit and `dirty` bit cleared

What if no Hardware Support?

What can the OS do if hardware does not have `use` bit (or `dirty` bit)?

- Can the OS "emulate" these bits?

Leading question:

- How can the OS get control (i.e., generate a trap) every time `use` bit should be set? (i.e., when a page is accessed?)

Problems with LRU-based Replacement

LRU does not consider frequency of accesses

- Is a page that has been accessed once in the past as likely to be accessed in the future as one that has been accessed N times?

Common workload problem:

- Scan (sequential read, never used again) of one large data region (larger than physical memory) flushes memory contents

Solution: Track frequency of accesses to page

Pure LFU (Least-frequently-used) replacement

- Problem: LFU can never forget pages from the far past

Combine LRU and LFU

LRU-K: Combines recency and frequency attributes

- Track K-th reference to page in past, replace page with oldest (LRU) K-th reference (or does not have K-th reference)
- History: Remembers access time of pages not now in memory
- Expensive to implement, LRU-2 used in databases

2Q: More efficient than LRU-2, similar performance

- Intuition: Instead of removing cold pages, only admit hot pages to main buffer
- `A1in` for short-term accesses, managed with FIFO
- `Am` for long-term accesses, managed with LRU
- On first page fault, page enters `A1in` queue
- On subsequent page faults, page enters `Am` queue
- `A1out`: pages not in buffer cache, but remembered

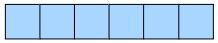
More Problems with LRU-based Replacement

Problematic workload for LRU

- Repeated scans of large memory region (larger than physical memory)

Example:

- 5 blocks of physical memory, 6 blocks of address space repeatedly scanned (ABCDEFABCDEFABCDEF...)
  - What happens with LRU?



Solution?

Policy Discovery for Real Systems

Page replacement policies not well documented

- Often change across versions of OS

Fingerprinting: automatic discovery of algorithms or policies (*e.g.* replacement policy, scheduling algorithm)

- Software that runs at user-level
- Requires no kernel modifications
- Portable across operating systems

Approach

- Probe the OS by performing sequence of operations (e.g., read/write)
- Time operations to see how long they take
- Use time to infer the policy the OS is employing

Dust

Research project in our group

- *"Exploiting Gray-Box Knowledge of Buffer-Cache Management", N. Burnett, J. Bent, A. Arpaci-Dusseau, R. Arpaci-Dusseau, USENIX"02*

Dust - Fingerprints file buffer cache policies

OS manages physical memory for two purposes

- file buffer cache
- virtual memory
- often same replacement policy for each (but not always)

Simpler to fingerprint file buffer cache policy

- read()
- seek()

Size of Memory

How can we infer the amount of physical memory used for buffer cache (or virtual memory)?

Idea for user-level fingerprinting process:

- Access some amount of data, `N` (i.e., bring it into physical memory)
- Re-access each page of data, timing how long each access takes
  - "Fast" access -->
  - "Slow" access -->
- If all pages appear in memory, increase `N` and repeat
- Stop when pages no longer all fit in memory
  - Implies `N > number of physical pages`

Replacement Policies

How can we infer replacement policy?

Assume policy uses combination of attributes:

- initial access order (FIFO)
- recency (LRU)
- frequency (LFU)

Algorithm

I. Move memory to known state by accessing `test` data such that each page has unique attributes

II. Cause part of `test` data to be evicted by reading in `eviction` data

III. Sample `test` data to determine cache state

- Read a block and time it
- Which blocks are still present determines policy of OS

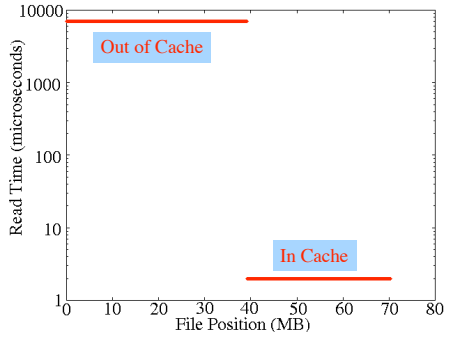
Repeat for confidence

Setting Initial Access Order


```
for ( 0 □ test_region_size/read_size) {
    read(read_size);
}
```

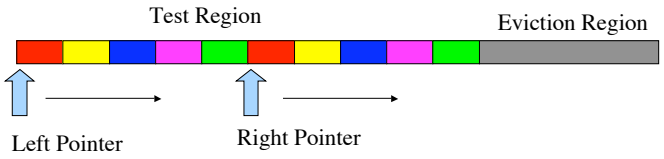
Detecting FIFO

Results from reading and timing `test` on FIFO simulator



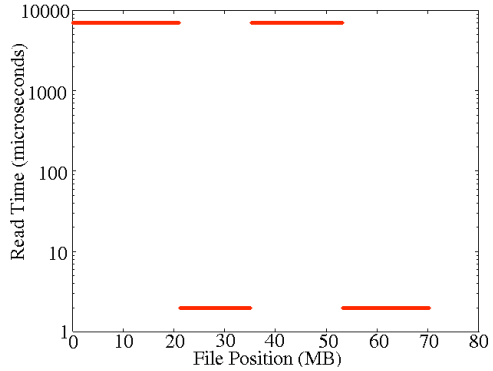
FIFO evicts the first half of `test` region

Setting Recency



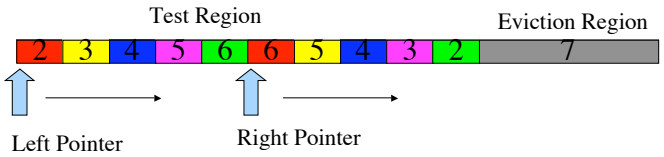

```
do_sequential_scan();
left = 0;  right = test_region_size/2;
for ( 0   test_region_size/read_size){
    seek(left); read(read_size);
    seek(right); read(read_size);
    right+=read_size; left+= read_size;
}
```

Detecting LRU



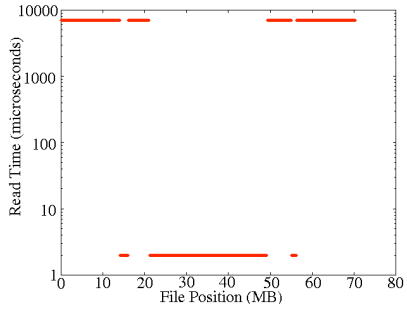
LRU evicts 1st and 3rd quarters of `test` region

Setting Frequency



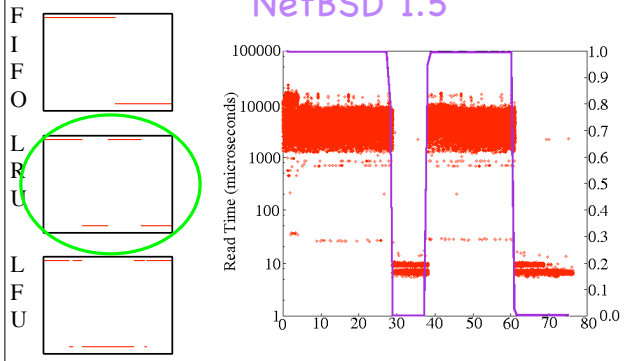

```
do_sequential_scan();
left = 0;  right = test_region_size/2;
left_count = 1; right_count = 5;
for ( 0   test_region_size/read_size)
    for (0   left_count) seek(left); read(read_size);
    for (0   right_count) seek(right); read(read_size);
    right+=read_size; left+= read_size;
    right_count++; left_count--;
```

Detecting LFU



LFU evicts outermost stripes of `test`
(Two stripes partially evicted)

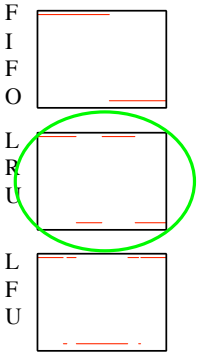
NetBSD 1.5

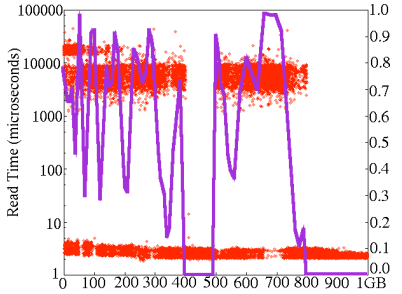


Noisier data on real systems due to disk performance variations

Conclusion: LRU

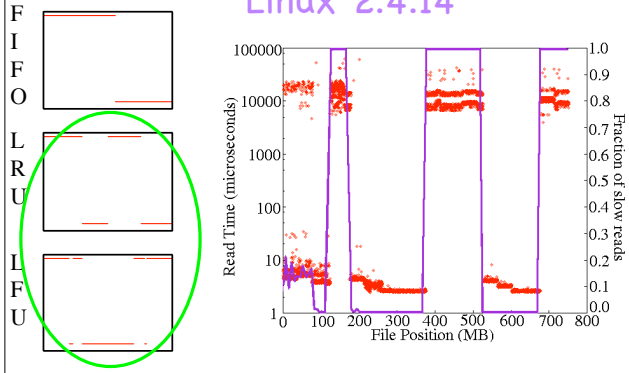
Linux 2.2.19





Very noisy but looks like LRU - actually clock

Linux 2.4.14

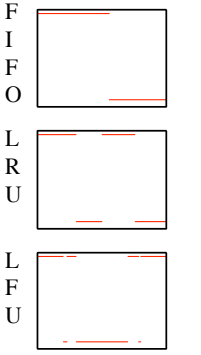


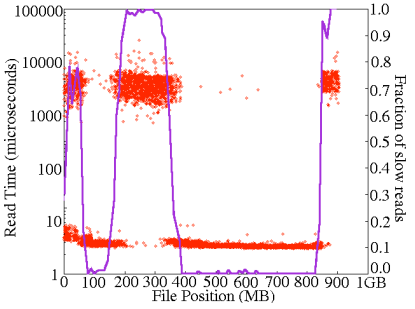
Low recency areas are evicted

Low frequency areas also evicted

Conclusion: LRU with page aging (similar to 2Q)

Solaris 2.7





Clock with some frequency component