Concurrent Systems

3D4 ⟷ CS2016

Operating Systems

Andrew Butterfield
ORI.G39, Andrew.Butterfield@scss.tcd.ie

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

with thanks to Mike Brady

Memory-management: the OS perspective

- What are we interested in from an OS perspective?
  - Page fault handling
- The goal of the operating system is to attempt to minimise the occurrence of page faults, which are expensive
  - Need to decide which physical memory page to use
  - Need to perform I/O
  - Another process needs to be scheduled to run during the I/O, resulting in a context switch
- We will look at several policies used by the OS to perform memory management
  - The key questions are:
    - When should a page be loaded?
    - Where (in physical memory) should it be loaded?

Fetch Policy (1)

- Demand Paging
  - Pages are loaded into memory only when they are first referenced
  - When a process is started, there will be a number of page faults in quick succession
  - Locality of reference: over a short period of time, most memory references are to a small number of pages
    - The pages references changed relatively slowly over time

Fetch Policy (2)

- Prepaging (performed in addition to demand paging)
  - Takes advantage of the characteristics of disk devices
    - Seek times and rotational latency
  - If a number of pages are contiguous in secondary storage, it is more efficient to load the contiguous pages, even though they haven't been referenced yet

Page Replacement Policy

- Which page frame should be used when a page fault occurs?
- A *page replacement policy* is required when there are no free page frames and we need to select one to replace

Page Replacement Policy – Basic Scheme

- Find a free frame
- If there was a free frame, then use it
- Otherwise if all the frames are used
  - select a victim frame using a page replacement policy
  - write the victim frame to disk (if needed) and update the page table entries accordingly
- Read the new page from disk into the selected page frame
- Allow the faulting process to resume

Page Replacement Policy – Optimal Algorithm

- Select the page for which the time until the next reference is furthest away…
- Obviously, we cannot implement this policy without prior knowledge of future events
- However, it is a useful policy against which more practical policies can be measured

Page Replacement Policy – Least Recently Used

- Replace the page in memory that was referenced furthest back in time
- Exploits locality of reference
- Near-optimal performance
- Disadvantage: very expensive to implement

LRU Implementation – Counter/Timestamp

- Associate a timestamp variable with each page table entry
- Implement a logical clock that is incremented on every memory access
- Whenever we access a memory page, we update the timestamp in the page table entry with the current logical clock value
- To replace a page we need to search the page tables to find the least recently used page
- Updates are cheap but searching for a replacement is very expensive

LRU Implementation – Stack

- Maintain a stack as a doubly-linked list of page table entries
- When a page is referenced, move it from its current position in the stack to the top of the stack
- Always replace the page at the bottom of the stack
- Updates are expensive but searching for a victim page is cheap

Page Replacement Policy – FIFO

- Replace pages in a round-robin order
- Very efficient to implement – maintain a pointer to the next page to be replaced
- Assumes that the page loaded into memory furthest back in time is also the page accessed furthest back in time
- Bad assumption: we might have a page that is accessed frequently for a long period of time
  - This page will be repeatedly loaded and unloaded

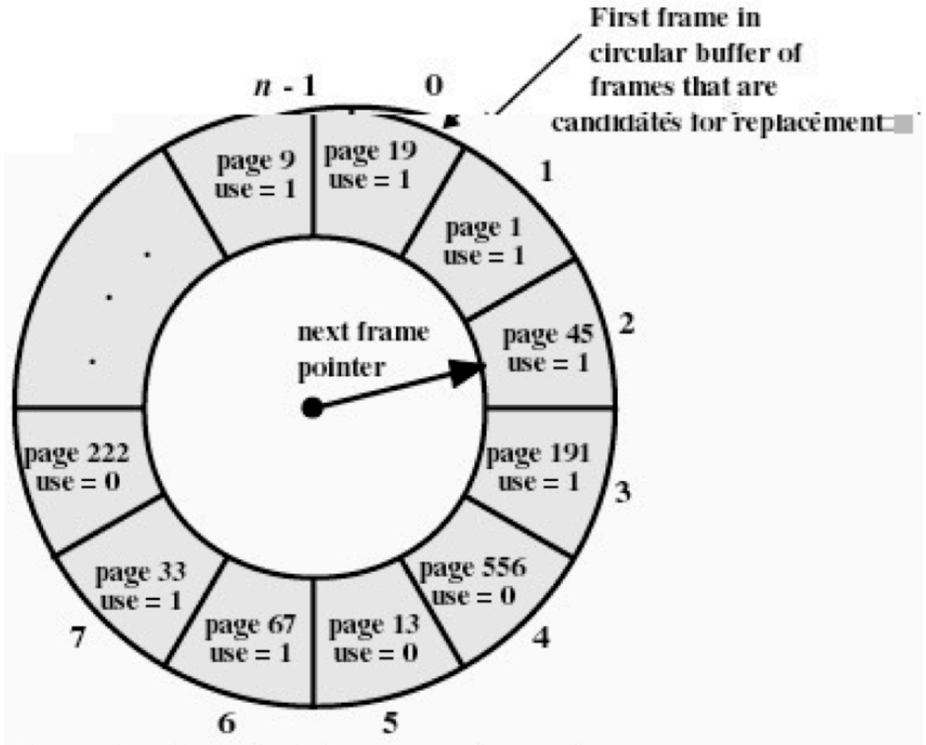
Page Replacement Policy – Clock Algorithm

- Performs better than FIFO but is not as inefficient to implement as LRU
- Conceptually, candidate pages for replacement are arranged in a circular list
- Each page has an associated use bit (also called reference bit)
  - Set to 1 when page is loaded or subsequently accessed

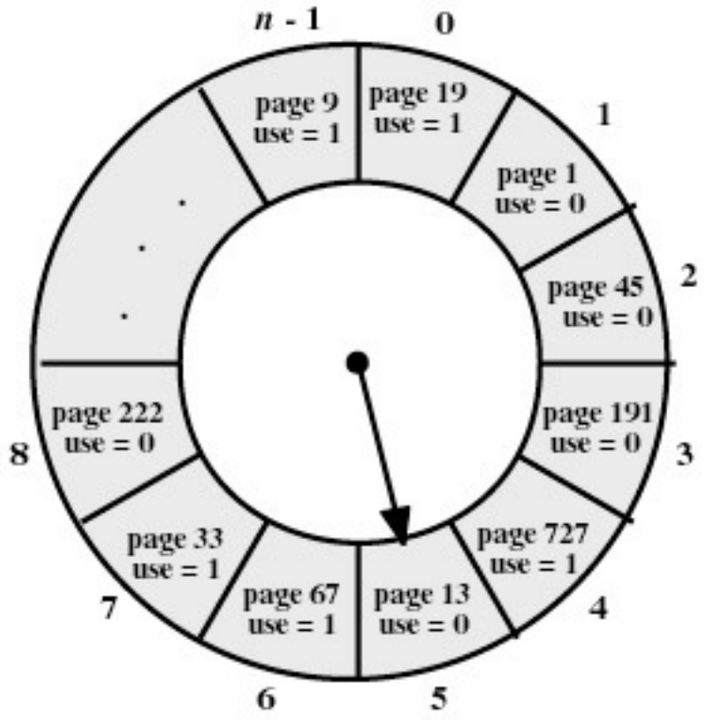
Clock Algorithm – Find a Candidate Page

- To find a page for replacement
  - Proceed around the list until a page with a use=0 is found
  - Every time we pass over a page with use=1, set use=0
  - If all pages have use=1, we will eventually arrive back where we started, but we will have set use=0
- Efficient to implement
- Similar to FIFO, except that any page with use=1 will be ignored, unless there is no page with use=0

Clock Algorithm



(a) State of buffer just prior to a page replacement



(b) State of buffer just after the next page replacement

Clock Algorithm with Modified Bit

- As well as use bits, we use pages' modified bits
- Algorithm:
  - First pass: look for use=0, modified=0, don't set use=0
  - Second pass: look for use=0, modified=1, set use=0
  - Third pass: As before, we are back at our starting position, but use will be set to 0 this time
- Why? It's cheaper to replace pages that don't need to be written first