08. Virtual Memory

9th ed: Ch. 8, 9

10th ed: Ch. 9, 10

Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging and the working set model
- To understand some page-replacement and allocation algorithms
- To be aware of problems of thrashing and Belady’s anomaly

Outline

- Virtual memory
- Page faults
- Page replacement
- Frame allocation

Outline

- Virtual memory
    - Virtual memory benefits
    - Virtual address space
- Page faults
- Page replacement
- Frame allocation

Virtual memory

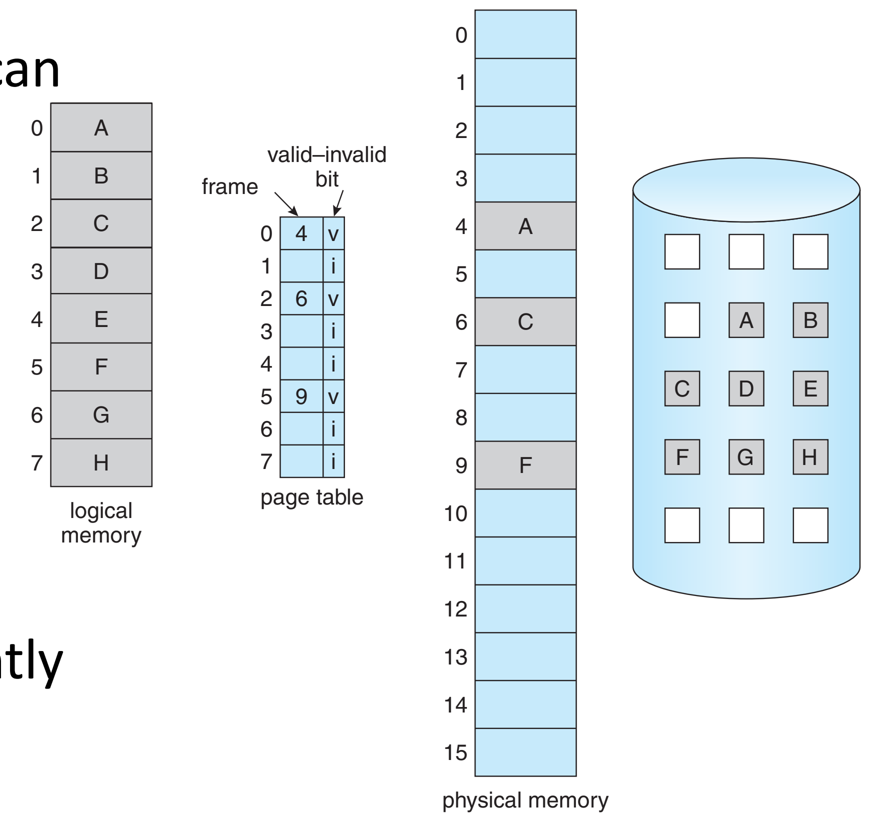
- Virtual addressing allows us to introduce the idea of virtual memory
- Already have valid or invalid page translations; introduce “non-resident” designation and put such pages on a non-volatile backing store
- Processes access non-resident memory just as if it were “the real thing”
- Separates program logical memory from physical memory, allowing logical address space to be much larger than physical address space
- Implemented via **demand paging** and **demand segmentation**

Virtual memory benefits

- Portability

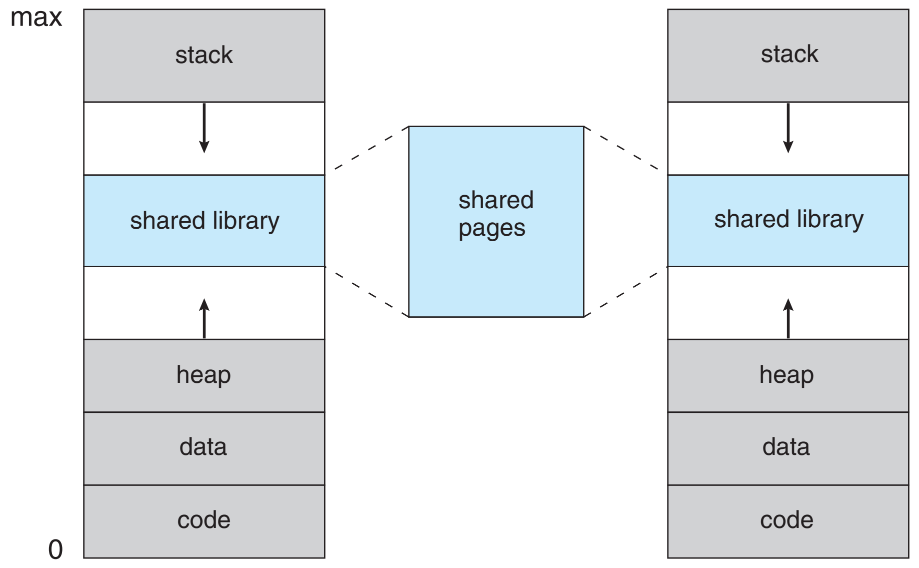
  Programs work regardless of how much physical memory, can be larger than physical memory, and can start executing before fully loaded

- Convenience
  - Less of the program needs to be in memory at once, thus potentially more efficient multi-programming, less IO loading/swapping program into memory, large sparse data-structures easily supported
- Efficiency
  - No need to waste (real) memory on code or data which isn't used (e.g., error handling or infrequently called routines)



Virtual address space

- Virtual address space gives the logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
- Physical memory organized in page frames
  - MMU must map logical to physical
- Usually stack starts at maximum logical address and grows “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between stack and heap is the hole
- No physical memory needed until heap or stack grows to a new page
  - Enables sparse address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
  - Shared memory by mapping pages read-write into virtual address space
  - Pages can be shared during fork(), speeding process creation

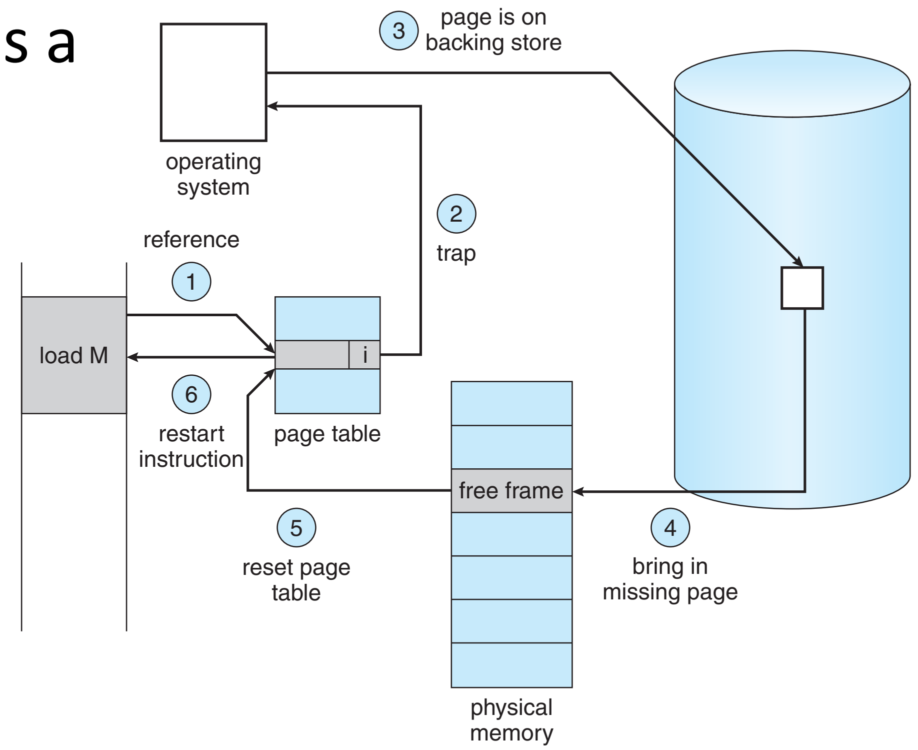


Outline

- Virtual memory
- Page faults
  - Instruction restart
  - Locality of reference
  - Demand paging
  - Optimisations
- Page replacement
- Frame allocation

Page faults

- When an invalid page is referenced, it causes a trap to the OS – a **page fault**
  - E.g., when referenced for the first time
- OS handles the trap by examining another table
  - If invalid memory reference, then abort
  - If valid but not resident, find a free frame and swap the page in
  - Entry is now marked valid as page is in memory
- After handing the fault, restart the instruction that caused the fault

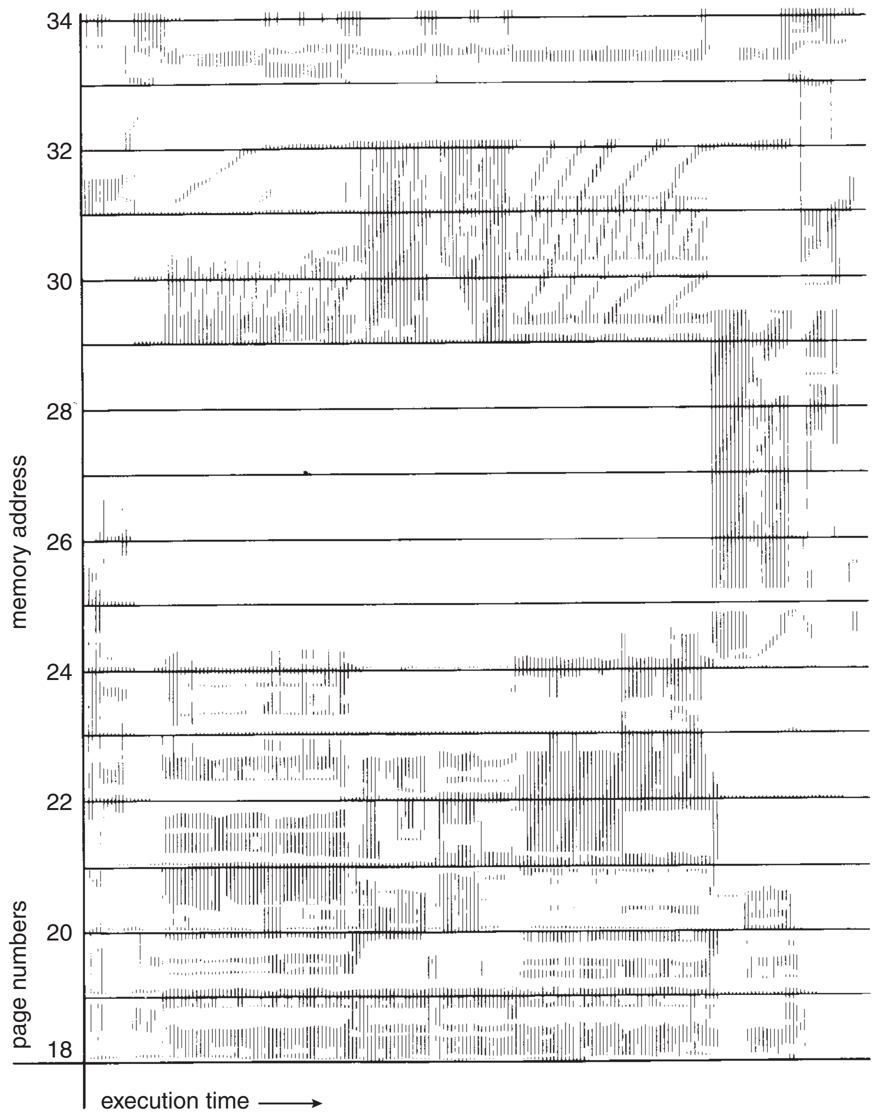


Instruction restart

- E.g., fetch and add two numbers from memory, and store the result back
  - Fetch and decode instruction (*add*), then fetch operands *A* and *B*, perform the addition, and store result to *C*
  - If store to *C* faults, need to handle the fault and then restart from the beginning (fetch and decode instruction, etc)
  - Locality of reference helps: unlikely to have multiple faults per instruction
- More complex: an instruction that could access several different locations
  - E.g., move a block of memory where source and destination can overlap, and either source or destination (or both) straddle a page boundary
  - As the instruction executes, the source might be modified – so it can't be restarted from scratch
  - Handle by, e.g., microcode for instruction strides across block, touching every page to ensure valid so no fault can occur
- **Double fault**: if the page fault handler itself triggers a fault – just give up…

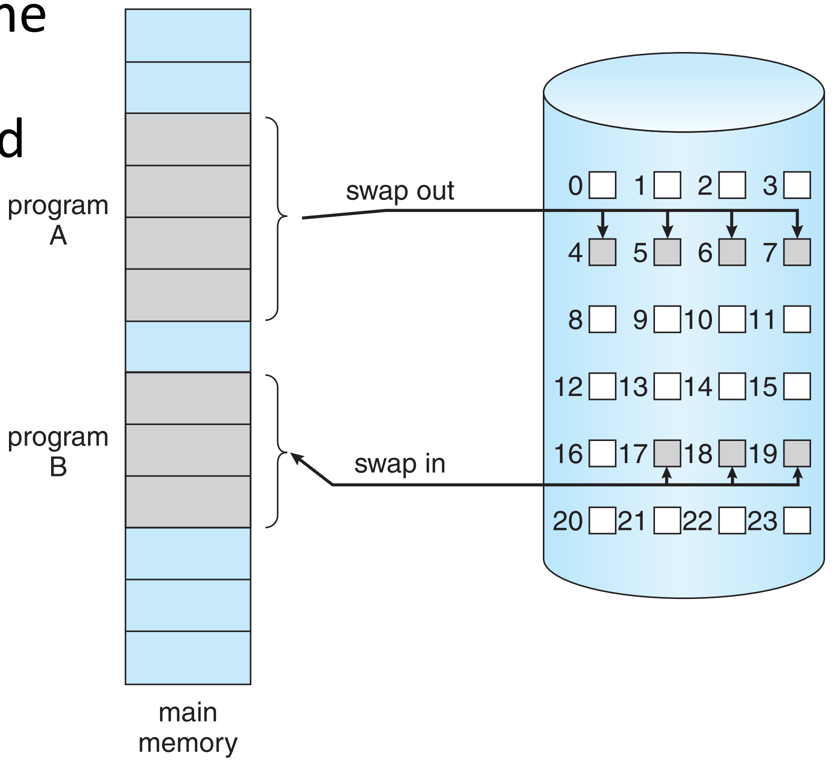
Locality of reference

- In a short time interval, the locations referenced by a process tend to group into a few regions of its address space
- E.g.,
    - Procedure being executed
    - Sub-procedures
    - Data access
    - Stack variables



Demand paging

- Could bring entire process into memory at load time, or bring pages into memory as needed
  - Reduces I/O and memory needed and response time
  - Supports more running processes
  - **Pure demand paging** starts with every page marked invalid
- Hardware support required
  - Page table with valid / invalid bit
  - Secondary memory (swap device with swap space)
  - Ability to restart instructions
- **Lazy swapper** (or **pager**) never swaps a page into memory unless page will be needed
  - But what to swap in and out?



Demand paging performance – worst case

1. Trap to the OS
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check the page reference was legal and find the page on disk
5. Issue a read from the disk into a free frame
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame
6. Reallocate CPU to another program
7. Receive an interrupt when disk I/O completes
8. Save the registers and process state for the other program
9. Determine that the interrupt was from the disk
10. Correct page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Demand paging performance

- Assume memory access time is 200ns, average page-fault service time 8ms, and page fault rate p
  - $0 \le p \le 1$: if $p = 0$, no page faults; if $p = 1$, every reference causes a fault
- **Effective Access Time** (**EAT**)
$$= (1 - p) \times 200\text{ns} + p \times 8\text{ ms}$$
$$= (1 - p) \times 200 + p \times 8{,}000{,}000 = 200 + 7{,}999{,}800\, p$$
- If one access in 1,000 causes a page fault, EAT $= 8.2\mu$secs — a 40x slowdown!
- For performance degradation below 10% require
$$220 \ge \text{EAT} = 200 + 7{,}999{,}800\, p$$
- Solving for p gives $p < 0.0000025$, i.e., less than one page fault per 400,000 accesses

Demand paging optimisations

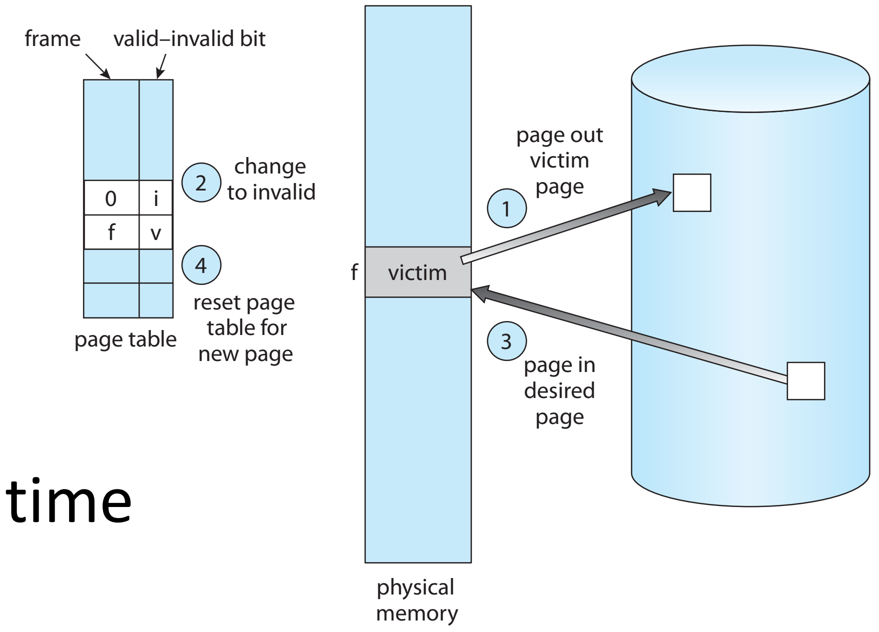
- Swap space I/O can be faster than file system I/O even on the same device
  - Allocate swap in larger chunks requiring less management than file system
  - Copy entire process image to swap space at process load time and then page in/out of swap space
- Demand page program from binary on disk – discard when freeing unmodified frame
- **Copy-on-Write** (**COW**)
  - Both parent and child processes initially share the same pages in memory
  - Only when a process actually modifies a shared page is the page copied
  - COW allows more efficient process creation as only modified pages are copied
- Allocate free pages from a pool of zero-fill-on-demand pages
  - Pool should always have free frames for fast demand page execution
  - Don’t want to have to free a frame as well as other processing on page fault
- *vfork* variation of *fork* has child created as copy-on-write address space of parent
  - Very efficient when the child just calls *exec*

Outline

- Virtual memory
- Page faults
- Page replacement
  - Algorithms
  - OPT, LRU
  - Counting algorithms
  - Page buffering algorithms
  - Performance
- Frame allocation

Page replacement

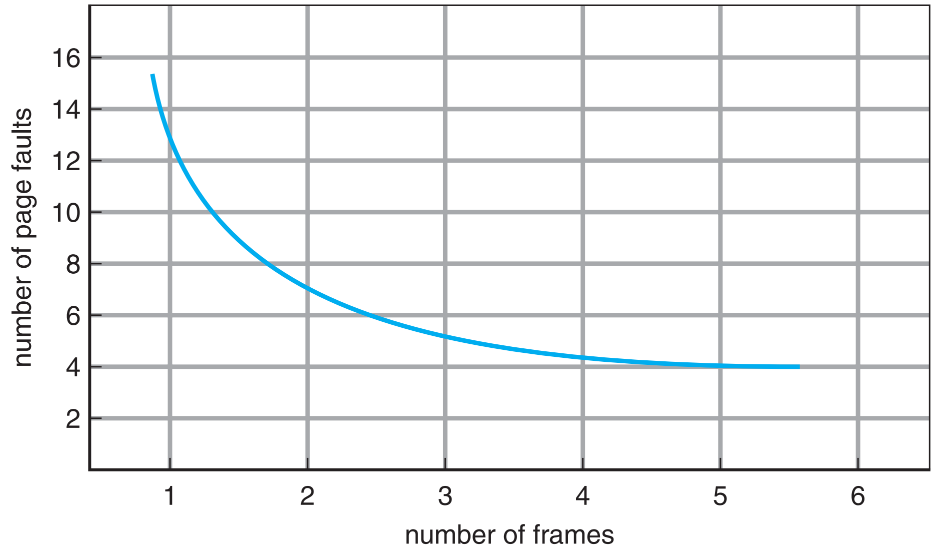
- Paging in from disk requires a free frame — but physical memory is limited
  - Either discard unused pages if total demand for pages exceeds physical memory size
  - Or swap out an entire process to free some frames
- Page fault handler must
  1. Locate the desired replacement page on disk
  2. Select a free frame for the incoming page:
     1. If there is a free frame use it, else select a victim page to free
     2. Write the victim page back to disk
     3. Mark it as invalid in its process' page tables
  3. Read desired page into the now free frame
  4. Restart the faulting process
- No free frames ~ doubles page fault service time



Page replacement algorithms

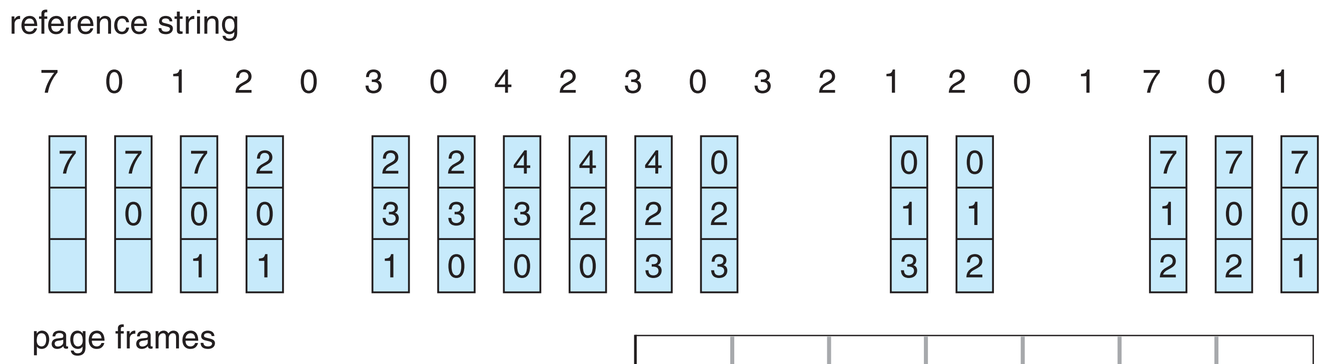
- Want the lowest page fault on both first and subsequent accesses
  - Evaluate using a sequence of page numbers, noting repeated access to same page does not trigger a fault

    7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
  - Assume three frames available
- Will look at three algorithms
  - First-In First-Out (FIFO)
  - Optimal (OPT)
  - Least Recently Used (LRU)

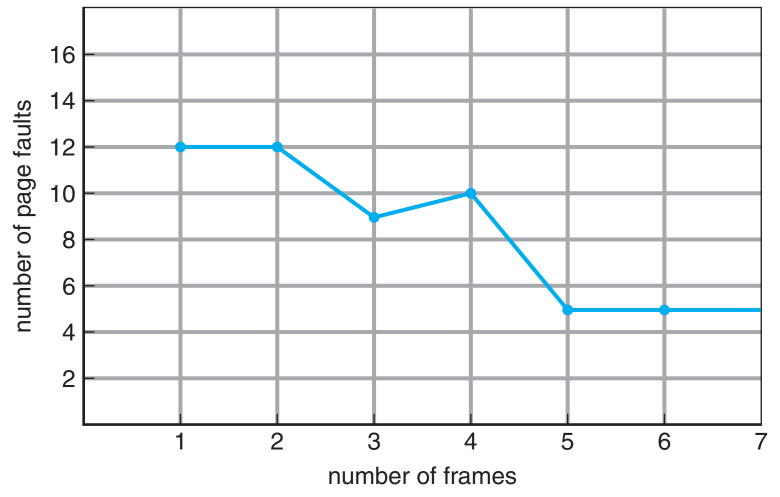


Page replacement algorithm: FIFO

- Simple FIFO queue for replacement gives 15 page faults

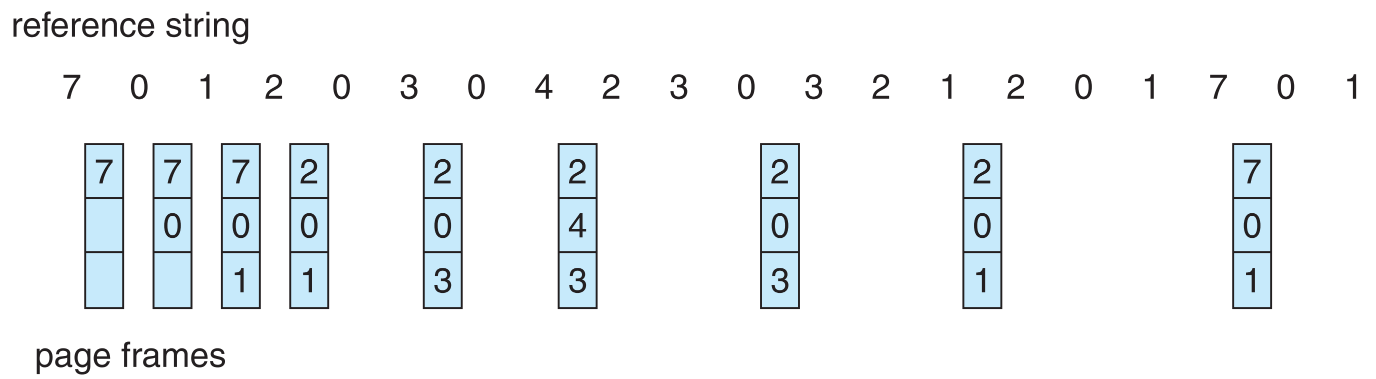


- Note that FIFO exhibits **Belady's Anomaly**
- As the number of frames **increases so can the number of page faults!**



Page replacement algorithm: OPT

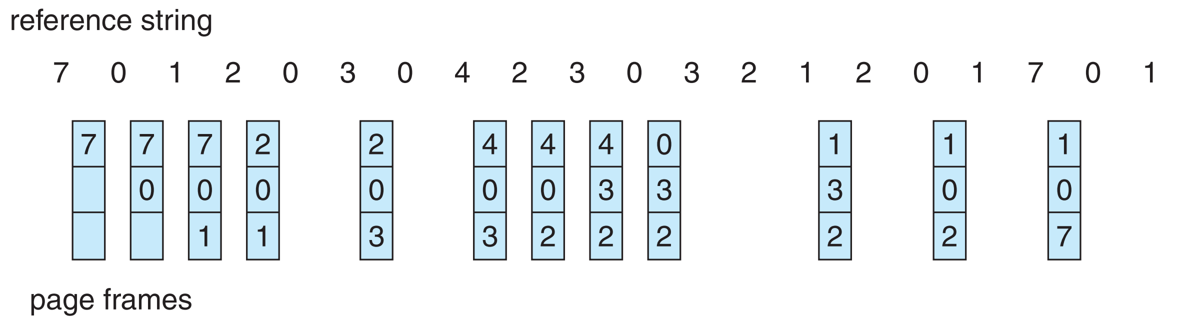
- Obvious: replace page that will not be used for the longest time



  - In this case, 9 is the best we can do
- Not obvious: how to build the oracle that knows the future
- Useful as a benchmark to measure how well your algorithm performs

Page replacement algorithm: LRU

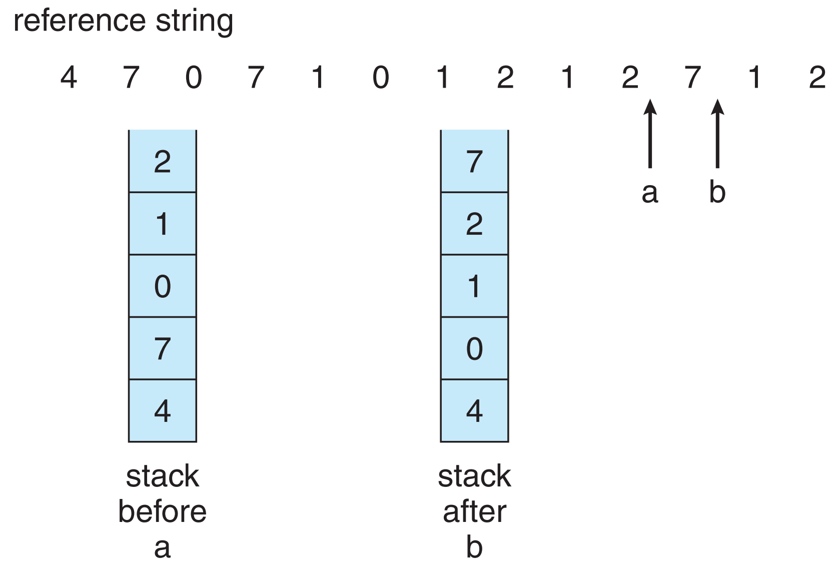
- Approximate OPT
  - Assume that the (recent) past is a good predictor of the future
  - Replace the page not used for the longest time



- Gives 12 faults – better than FIFO but worse than OPT
  - Generally good, frequently used – but how to implement?
  - Note both LRU and OPT are **stack algorithms** so don't have Belady's Anomaly

LRU implementation

- Counter implementation
  - Each PTE holds clock value, updated when page referenced through this PTE
  - Replace page with smallest counter value
  - Requires search through table, as well as memory write on every access
- Stack implementation
  - Maintain doubly-linked stack of page numbers
  - When page is referenced, move it to the top
  - Requires up to six pointers to be changed
  - Tail always points at the replacement



Approximating LRU

- Use a **reference bit** in the PTE, initially 0 and set to 1 when page touched
- **Not Recently Used** replacement
  - Periodically (every 20ms) clear reference bits
  - Victimise pages according to reference (and dirty) bits
  - Better: use an 8 bit value, shift bit in from the left
  - Maintains history for last 8 clock sweeps

Referenced?	Dirty?	Comment
no	no	best type of page to evict
no	yes	next best (needs writeback)
yes	no	probably code in use
yes	yes	bad choice of victim

- **Second-chance** (**Clock**) algorithm
  - Store pages in queue as per FIFO, often with a circular queue and a current pointer
  - Discard current if reference bit is 0 else reset reference bit (second chance) and increment current
  - Guaranteed to terminate after at most one cycle; devolves into a FIFO if all pages are referenced
- Can emulate reference bit (and dirty bit) if no hardware support
  - Mark page *no access* to clear reference bit
  - Reference causes a trap – update PTE, and resume
  - Check permissions to check if referenced

Counting algorithms

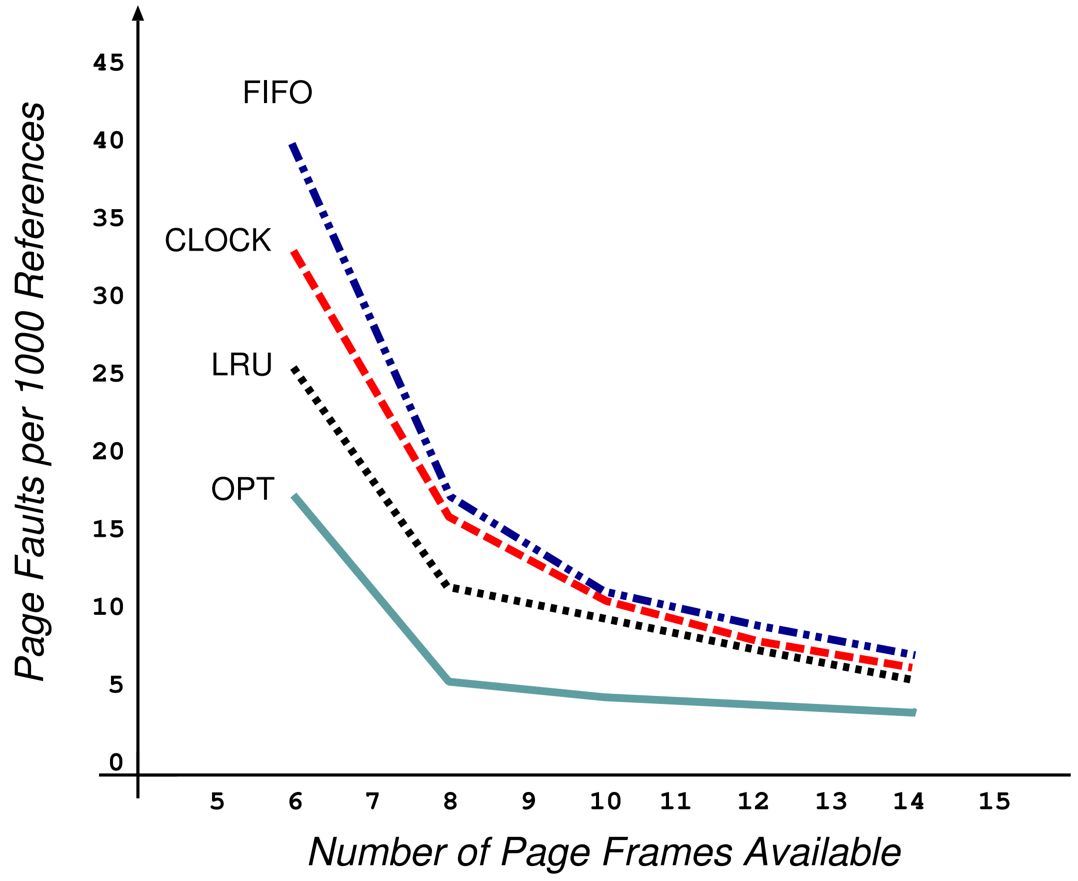
- Keep a count of the number of references to each page
- **Least Frequently Used** (**LFU**)
  - Replace page with smallest count
  - Takes no time information into account
  - Page can stick in memory from initialisation
  - Need to periodically decrement counts
- **Most Frequently Used** (**MFU**)
  - Replace highest count page
  - Low count indicates recently brought in
- Neither is common: expensive and don’t emulate OPT well

Page buffering algorithms

- Keep a minimum sized pool of free frames, always available
  - Read page into free frame before selecting victim and adding to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected
- Alternatively, stop having the OS guess about future page access
  - Applications may have better knowledge, e.g., databases
  - OS can give raw access to the disk, getting out of the way of the applications

Page replacement performance comparison

- Compare page-fault rate against number of physical frames
  - Pseudo-local reference string
  - Note offset x origin
- Seek to minimise area under curve
  - Getting the frame allocation right has major impact
  - Much more than which page replacement algorithm you use!



Outline

- Virtual memory
- Page faults
- Page replacement
- Frame allocation
  - Global vs local
  - Thrashing
  - Working set

Frame allocation

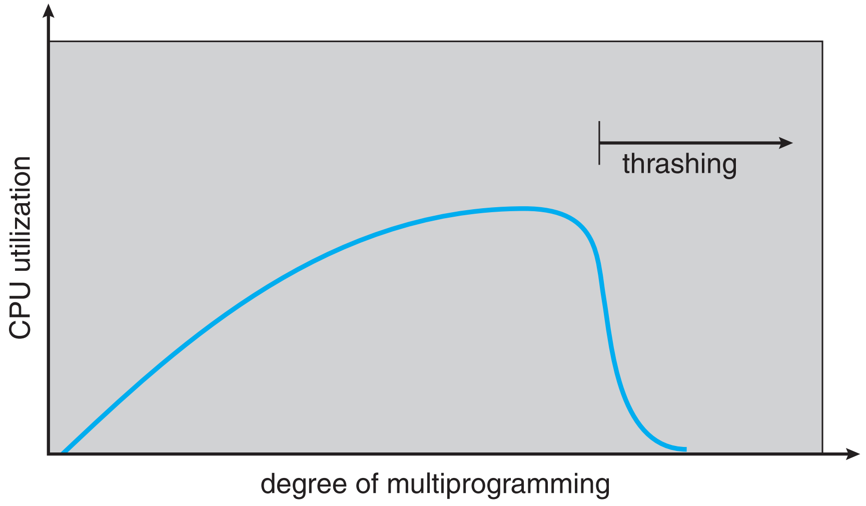
- Need an allocation policy to determine how to distribute frames
  - After reserving a fraction of physical memory per-process and for OS code/data
- Objective: Fairness (or proportional fairness)?
  - E.g. divide m frames between n processes as m/n, remainder in free pool
  - E.g. divide frames in proportion to size of process (i.e. number of pages used)
- Objective: Minimize system-wide page-fault rate?
  - E.g. allocate all memory to few processes
- Objective: Maximize level of multiprogramming?
  - E.g. allocate minimum memory to many processes

Global / Local allocation

- Most replacement schemes are **global**: any page could be a victim
  - Process execution time can vary greatly but greater throughput so more common
  - Allocation policy implicitly enforced during page-in: allocation only succeeds if policy agrees
  - Process cannot control its own page fault rate: performance can depend entirely on what other processes do
- E.g., given 64 frames and 5 processes, each gets 12 with four left over
  - When a process next faults after another process has died, it will allocate a frame
  - Eventually all will be allocated and a newly arriving process will need to steal some pages back from the existing allocations
- Alternatively, **local** replacement
  - Each process selects from only its own set of allocated frames
  - More consistent per-process performance but possibly underutilised memory

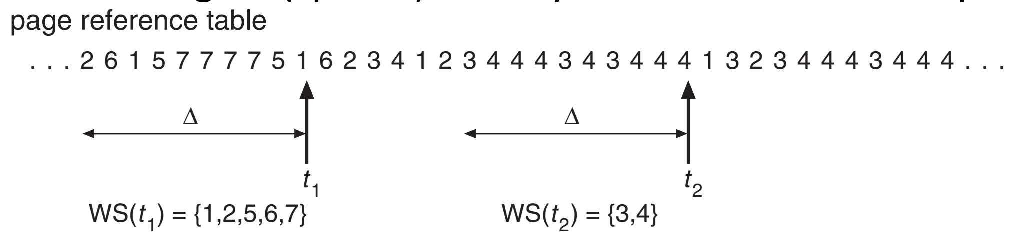
Thrashing

- A process without “enough” pages has high page-fault rate
  - Page fault to get page, replacing existing frame
  - But quickly need replaced frame back
- Cascading failure
  - Time wasted handling page faults leads to low CPU utilisation
  - Low CPU utilisation triggers OS think to increase degree of multiprogramming
  - This adds another process added to the system, increasing memory pressure
  - Collapse
- Why does demand paging work? Locality
  - Process migrates from one locality to another
  - Localities may overlap
- Thrashing occurs when size of locality > total memory
  - Limit effects by using local or priority page replacement



Working set

- Avoid thrashing by considering the **working set**
  - Those pages required at the same time for a process to make progress
  - Varies between processes and during execution
  - Assume process shifts phases but gets (spatial) locality of reference in each phase



- E.g., consider a window Δ of a fixed number of page references, say 10,000 instructions
  - Working set of process P_i is WSS_i, total number of pages referenced in the most recent window
  - Δ too small will not encompass entire locality
  - Δ too large will encompass several localities (entire program)
- Demand, $D = \sum_i WSS_i$, approximation of locality
  - Thrashing occurs if $D > m$, number of frames, in which case suspend/swap out a process
  - Approximate with interval timer and a reference bit: page in working set if one reference bit is set
  - **Pre-paging**: bring in working set pages when (re-)starting a process

Summary

- Virtual memory
  - Virtual memory benefits
  - Virtual address space
- Page faults
  - Instruction restart
  - Locality of reference
  - Demand paging
  - Optimisations
- Page replacement
  - Algorithms
  - OPT, LRU
  - Counting algorithms
  - Page buffering algorithms
  - Performance
- Frame allocation
  - Global vs local
  - Thrashing
  - Working set