CSCI [4|6] 730
Operating Systems

Virtual Memory

Virtual Memory Questions?

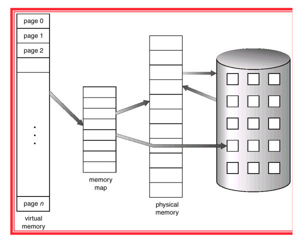
- What is virtual memory and when is it useful?
- What is demand paging?
- What pages should be
  - » resident in memory, and
  - » which should be replaced?
- What is the working set model?

Operating System's Goals

- Support processes when there is not enough physical memory
  - » Single process with very large address space
  - » Multiple processes with combined address spaces
- User code should be independent of amount of physical memory
  - » Correctness, if not performance

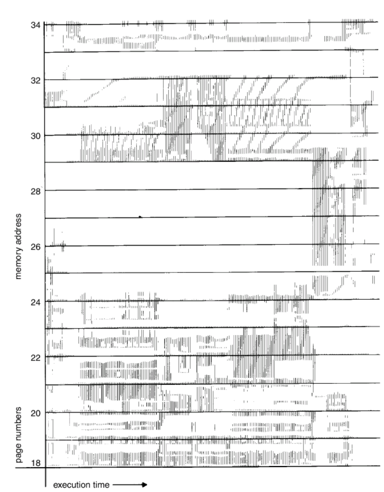
The Illusion: "*Virtual*" Memory

- OS provides an *illusion* of more memory than is physically available:
  - » Large logical space but really
  - » small physical memory
- Why does this work?
  - » Only part of the program needs to be in memory (at a particular time) for execution
  - » Relies on key properties of user processes
    - – workload and
    - – machine architecture (hardware)



The Million Dollar Question?

- How do the OS decide what is in "main" memory and what is on disk?
- How can we decide?
  - » Memory Access Patterns

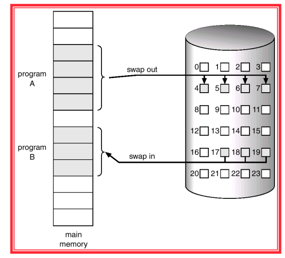


Observations: Memory Access Patterns

- Sequential memory accesses of a process are predictable and tend to have *locality of reference*:
  - » Spatial: reference memory addresses *near* previously referenced addresses (in memory)
  - » Temporal: reference memory addresses that have referenced in the past
- Processes spend majority of time in small portion of code
  - » Estimate: 90% of time in 10% of code
- Implication:
  - » Process only uses small amount of address space at any moment
  - » Only small amount of address space must be resident in physical memory

Approach: Demand Paging

- Bring in pages into memory only when needed
  - » Less memory
  - » Less I/O
  - » Faster response time?
- Process viewed as a sequence of page accesses rather than contiguous address space

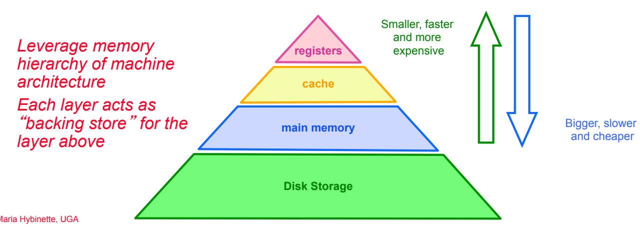


Virtual Memory Approach: Intuition

- Idea: OS keeps unreferenced pages on disk
  - » Slower, cheaper backing store than memory
- Process can run when not all pages are loaded into main memory
- OS and hardware cooperate to provide illusion of large disk as fast as main memory
  - » Same behavior as if all of address space in main memory
  - » Hopefully have similar performance
- Requirements:
  - » OS must have mechanism to identify location of each page in address space in memory or on disk
  - » OS must have policy for determining which pages live in memory and which (remain) on disk

Virtual Address Space Mechanisms

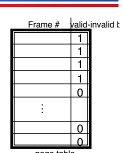
- Each page in virtual address space maps to one of three locations:
  - » Physical main memory: Small, fast, expensive
  - » Disk (backing store): Large, slow, cheap
  - » Nothing (error): Free

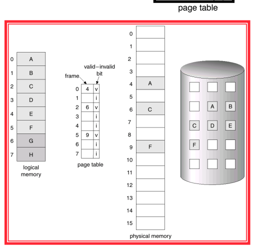


Virtual Address Space Mechanisms

Extend page tables with an extra bit to indicate whether it is in memory or on disk (a resident bit):

- `valid (or invalid)`
- Page in memory: valid bit set in page table entry (PTE)
- Page out to disk: `valid` bit cleared (`invalid`)
  - – PTE points to block on disk
  - – Causes trap into OS when page is referenced
  - – Trap: page fault



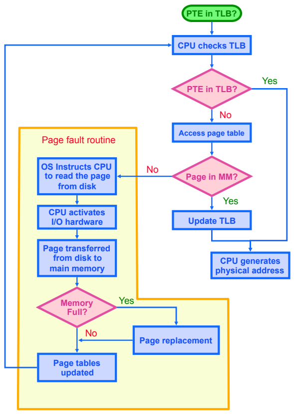


Virtual Memory Mechanisms (cont)

The TLB factor: Hardware and OS cooperate to translate addresses

- First, hardware checks TLB for virtual address
  - » TLB hit: Address translation is done; page in physical memory
  - » TLB miss:
    - – Hardware or OS walk page tables
    - – If PTE designates page is valid, then page in physical memory
- Main Memory Miss: Not in main memory: Page fault (i.e., `invalid`)
  - » Trap into OS (not handled by hardware)
  - » [if memory is full)] OS selects victim page in memory to replace
    - – Write victim page out to disk if modified (add `dirty` bit to PTE)
  - » OS reads referenced page from disk into memory
  - » Page table is updated, `valid` bit is set
  - » Process continues execution

Flow of “Paging” Operations



Virtual Memory Policies

- OS needs to decide on policies on page faults concerning:
  - » Page selection (When to bring in)
    - – When should a page (or pages) on disk be brought into memory?
    - – Two cases
      - When process starts, code pages begin on disk
      - As process runs, code and data pages may be moved to disk
  - » Page replacement (What to replace)
    - – Which resident page (or pages) in memory should be thrown out to disk?
- Goal: Minimize number of page faults
  - » Page faults require milliseconds to handle (reading from disk)
  - » Implication: Plenty of time for OS to make good decision

The *When*: Page *Selection*

- When should a page be brought from disk into memory?
- Request paging: User specifies which pages are needed for process
  - » Problems:
    - – Manage memory by hand
    - – Users do not always know future references
    - – Users are not impartial (and infact they may be wrong)

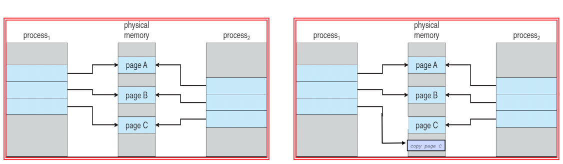
- Demand paging: Load page only when page fault occurs
  - » Intuition: Wait until page must absolutely be in memory
  - » When process starts: No pages are loaded in memory
  - » Advantage: Less work for user
  - » Disadvantage: Pay cost of page fault for every newly accessed page

Page *Selection* Continued

- Prepaging (anticipatory, prefetching): OS loads page into memory *before* page is referenced
  - » OS predicts future accesses (oracle) and brings pages into memory ahead of time
    - – How?
    - – Works well for some access patterns (e.g., sequential)
  - » Advantages: May avoid page faults
  - » Problems? :
- Hints: Combine ***demand*** or ***prepaging*** with user-supplied hints about page references
  - » User specifies: may need page in future, don’t need this page anymore, or sequential access pattern, ...
  - » Example: `madvise()` in Unix (1994 4.4 BSD UNIX)

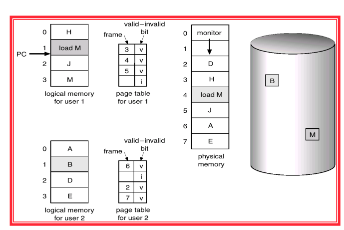
Virtual Page Optimizations

- Copy-on-Write: on process creation allow parent and child to share the same page in *memory until one modifies* the page.



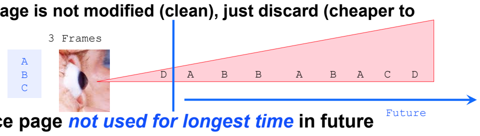
What happens if there is no free frame?

- Page replacement
  - » find some page in memory, that is not really in use, and swap it out.
- Observation: Same page may be brought into memory several times (so try to keep that one in memory)



Page *Replacement Strategies*

- Which page in main memory should selected as victim?
  - » Write out victim page to disk if modified (dirty bit set)
  - » If victim page is not modified (clean), just discard (cheaper to replace)



- **OPT**: Replace page *not used for longest time* in future
  - » **Advantage**: Guaranteed to minimize number of page faults
  - » **Disadvantage**: Requires that OS predict the future
    - – Not practical, but is good to use comparison (best you can do)
- **Random**: Replace any page at random
  - » **Advantage**: Easy to implement
  - » Surprise?: Works okay when memory is not severely over-committed (recall lottery scheduling, random is not too shabby, in many areas)

Page Replacement Continued

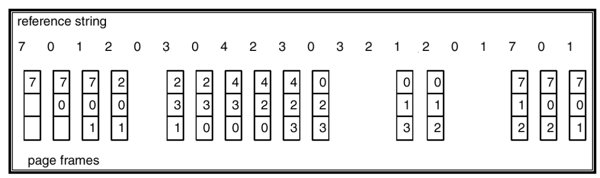
- **FIFO**: Replace page that has been in memory the longest
  - » **Intuition**: First referenced long time ago, done with it now
  - » **Advantages**:
    - – Fair: All pages receive equal residency
    - – Easy to implement (circular buffer)
  - » **Disadvantage**: Some pages may always be needed
- **LRU**: Replace page *not used* for longest time in past
  - » **Intuition**: Use the past to predict the future
  - » **Advantages**:
    - – With locality, LRU approximates OPT (but look backwards)
  - » **Disadvantages**:
    - – Harder to implement, must track which pages have been accessed
    - – Does not handle all workloads well

MFR, LFU

How to Evaluate Page Replacement Algorithms?

- **Want:** lowest page-fault rate (least #misses)
- **Idea**: Keep track of memory references – test with particular string of memory references and count page faults (based on real data or generated)
- **Algorithm**: Convert address to page location
  - » Example: Assume 100 bytes per page and
    - – **Step 1**: Assume the address sequence:
      - 0100, 0210, 0250, 0300, 0350, 0380, 0400, 0160, 0250, 0505, 0100, 0110, 0230, 0350, 0450, 0450, 0500, 0500
    - – **Step 2**: Convert address to a page reference string:
      - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.
    - – **Step 3**: Count page faults.

Example: Counting Faults of FIFO Page Replacement Algorithm



First IN is the one that is first out

- **3 Frames are available**
- **FIFO**: Replace page that has been in memory the longest
- Count page faults ?

Page Replacement Example

Page reference string: A B C A B D A D B C B
Three pages of physical memory

	OPT	FIFO	LRU
ABC			
A			
B			
→ D			
A			
D			
B			
C			
B			

Page Replacement *Example*

Page reference string: A B C A B D A D B C B
Three pages of physical memory

	OPT [5]	FIFO [7]	LRU [5]
ABC	A B C	A B C	A B C
A			
B			
→ D	A B D	D B C	A B D
A		D A C	
D			
B		D A B	
C	B	C A B	C B D
B			

Page Replacement: Adding *More* Memory

- Add more physical memory, what happens to performance?
  - » Ideally the numbers of page faults should should *decrease* as the number of available frames *increases*
  - » 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.
  - » If 1 page frame : Number of page faults? (lots)
    - – 12 page faults, one fault for every page
  - » If 12 frames : Number of page faults? (fewer)
    - – 5 page faults

First-In-First-Out (FIFO) Algorithm: Add Memory (3 Frames to 4 Frames)

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

1	2	3
4	1	2
5	3	4

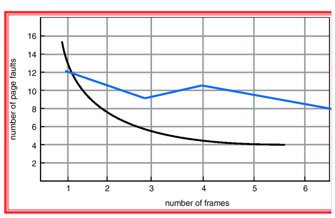
- 4 frames

1	2	3	4
5	1	2	3
4	5		

- FIFO Replacement – Belady's Anomaly
  - » Violates the Principle: More frames ⇒ less page faults
  - » 9 PF -> 10 PF (more page faults as we increase memory)
  - » There is some string that have more page faults)

Summary : Page Replacement: Add memory

- Add more physical memory, what happens to performance?
  - » Ideally the numbers of page faults should should decrease as number of available frames increases
  - » 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.
  - » If 1 page frame : 12 faults every access is fault
  - » If 3 page frame: 9 faults
  - » If 4 page frame: 10 faults
  - » If 12 frames : 5 faults



Page Replacement Comparison

- Add more physical memory, what happens to performance?
  - » LRU, OPT: Add more memory, guaranteed to have fewer (or same number of) page faults
    - – Smaller memory sizes are guaranteed to contain a subset of larger memory sizes
  - » FIFO: Add more memory, usually have fewer page faults
    - – Belady's anomaly: But may actually have more page faults!

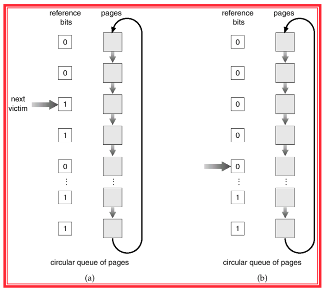
Implementing LRU

- Software Perfect LRU (Stack)
  - » OS maintains ordered list of physical pages by reference time
  - » When page is referenced: Move page to front of list (top) (slow, search is n to find page).
  - » When need victim: Pick page at back of list (bottom) (fast (1)
  - » Trade-off: Slow on memory reference (find it), fast on replacement
- Hardware Perfect LRU
  - » Associate register with each page (fast access)
  - » When page is referenced: Store system clock in register (fast)
  - » When need victim: Scan through registers to find *oldest* clock (slow)
  - » 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 it more…
  - » Goal: Find an old page, but not necessarily the very oldest (just old enough)

Clock or *Second* Chance Algorithm

- Hardware (use a reference bit)
  - » Keep `use` (or `reference`) bit for each page frame initialized to 0.
  - » When page is referenced: set `use` bit (1), making it less likely to be replaced.
- Operating System
  - » Page replacement: Look for page with `use` bit cleared (0) (has not been referenced for a while)
  - » Implementation:
    - – Keep pointer to last examined page frame
    - – Traverse pages in circular buffer
    - – Clear `use` bits while searching for replacement
    - – Stop when find page with already cleared `use` bit, replace this page

Clock Algorithm Example



- **Worst Case:**
  - » **All bits are set -> FIFO (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 (multiple zeros)**
- **Use a Two-handed clock**
  - » **Intuition (problem of 1 handed clock)**
    - **If it takes long time for clock hand to sweep through pages, then all use bits might be set (all are 1s)**
    - **Traditional clock cannot differentiate between usage of different pages (only between 1s and 0s).**
  - » **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 a software byte (to keep a bit mask)**
  - » **Intuition: Keep track of history when last used**
- **Implementation: Reference bit**
  - » **With each page associate a bit, initially = 0**
  - » **When page is referenced bit set to 1.**
  - » **Keep a *history* of reference bit in an (8 bits) byte:**
    - **Shift reference bit for each page to high order bit, and other bits right one bit.**
    - **11000100 (more recently used than below)**
    - **01110111**

More Clock Extensions (R/W)

- **Use `dirty` bit to give preference to dirty pages (to stay)**
  - » **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**

0, 0	Not recently used, not modified	Best to replace
0, 1	Not recently used, but modified	Needs to be written out
1, 0	Recently used, not modified	Probably used again soon
1, 1	Recently used and modified	Probably used again soon and need to be written out

Problems with LRU-based Replacement

- **Locality of reference:**
  - » **Same pages referred frequently (warm pages)**
  - » **Example:** 2, **1**, **3**, 2, **4**, 2, **4**, **1**, **5**, **6**, 2, …
- **LRU takes advantage of this!**
- **Leading question:**
  - » **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?**

Problems with LRU-based Replacement

- **Example:** 2, **1**, **3**, 2, **4**, 2, **4**, **1**, **5**, **6**, 2, …
- **Problem:**
  - » **Dislodges warm pages if a long sequence of one time page references occur.**
    - **In the above ex, page 2 may get dislodged by the access pattern …, 4, 1, 5, 6,**
  - » **LRU does not consider *frequency* of accesses**
- **Solution: Track frequency of accesses to page**
  - » **Pure LFU (Least-frequently-used) replacement**
- **Problem: but LFU can never forget pages from the far past… (so we need to add aging to the algorithm….)**

Questions

- How to allocate memory *across* competing processes?
- What is thrashing? What is a working set?
- How to ensure working set of all processes fit?

Allocating Memory across Processes

- **Problem**:
  - » 2 processes and 25 free frames how are these frames divided up between processes?
- **Three General Approaches:**
  - » Global Replacement
  - » Per-Process Replacement
  - » Per-User Replacement (set of processes linked to a user)

Global Replacement

- Global replacement
  - » Pages from all processes lumped into single replacement pool
  - » Each process competes with other processes for frames
  - » Advantages:
    - – Flexibility of allocation
    - – Minimize total number of page faults
  - » Disadvantages:
    - – One memory-intensive process can hog memory, hurt all other processes (not fair)
    - – Paging behavior of one process depends on the behavior of other processes

Per-process replacement

- Per-process free pool of pages:
  - » Equal, Fixed Allocation: Fixed number of pages per process
    - – 100 frames and 5 processes, give each 20 pages.
    - – Fixed fraction of physical memory
  - » Proportional Allocation:
    - – Proportional to size of address space of a process.
    - – Adjust size allocated if a process have higher priority
- Page fault in one process only replaces frame of that process
- Advantage: Relieves interference from other processes
- Disadvantage: Potentially inefficient allocation of resources

Per-User Replacement

- Advantages: Users running more processes cannot hog memory
- Disadvantage: Inefficient allocation

Over Committing Memory

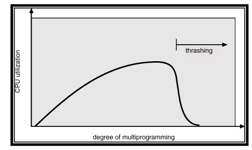
- When does the Virtual Memory illusion break?
- Example:
  - » Set of processes *frequently* referencing 33 important pages - more than the memory available (then you are stuck with always replacing a page that is frequently referenced).
    - – Physical memory can fit 32 pages
- What happens?
  - » System Repeat Cycle:
    - – Reference page not in memory
    - – Replace a page in memory with newly referenced page
    - – Replace another page right away again, since all its pages are in active use…

Thrashing

- **Thrashing:**
  - » **Definition: Spends more time paging than execution, i.e. system reading and writing pages instead of executing useful instructions**
  - » **Observation - Global replacement algorithm aggravates.**
  - » **Symptom: Average memory access time equals to disk access time**
    - – **Breaks the virtual memory illusion because memory appears as slow as disk rather than disk appearing fast as memory (system is reading/writing instead of executing)**
    - – **Memory appears as slow as disk, instead of disk appearing as fast as memory**
  - » **Processes execute less – system admits more processes -> thrashing gets worse**

System does not know it is thrashing

- **If a process does not have "enough" pages, the page-fault rate is very high.**
  - » **low CPU utilization.**
  - » **operating system thinks that it needs to increase the degree of multiprogramming.**
  - » **another process added to the system**
- **Why the CPU utilization decreases:**
  - » **Suppose a process need more frames, starts faulting, removing frames from others, in turn making the other processes fault**
  - » **Processes queue up for the paging device, CPU decreases**
  - » **OS add processes that immediately need new frames further taking away pages from running processes**



Thrashing: Solutions

- **Limit thrashing by using a local replacement**
  - » **Process does not steal frames from other and cause others to thrash**
  - » **Average service time for a page fault can still increase…**
- **Admission Control:**
  - » **Determine of much memory each process needs**
  - » **Long-term scheduling policy:**
    - – **Run only processes whose memory requirement can be satisfied**
- **What if memory requirement of one process is too high?**
  - » **Observation: a process moves through different ``localities'' through out is lifetime**
    - – **Locality: Set of pages that are actively used together**
  - » **Solution: Idea: Amortize page allocated so that a process get enough page for its current locality….**

Motivation for Solution

- **Thrashing cannot be fixed with better replacement policies**
  - » **Page replacement policies do not indicate that a page must be kept in memory**
  - » **Only show which pages are better than others to replace**
- **Student's analogy to thrashing: Too many courses**
  - » **Solution: Drop a course (focus on other remaning courses)**
- **OS solution: Admission control**
  - » **Determine how much memory each process needs**
  - » **Long-term scheduling policy**
    - – **Run only those processes whose memory requirements can be satisfied**
  - » **What if memory needs of one process are too large?**

Working Set

- **Informal definition**
  - » **Collection of pages the process is referencing frequently**
  - » **Collection of pages that must be resident to avoid thrashing**
- **Formal definition**
  - » **Assume locality; use recent past to predict future**
  - » **Pages referenced by process in last T seconds of execution**
  - » **Working set changes slowly over time**
- **Example (figure out number of frames needed by inspecting the past using a window based approach)**

$\Delta = 8$ Time →

A A B C B B B C | D C D E B B E E D F B F D B | B E D B

A B C   B D E F

- Balance Set -

- **Motivation: Process should not be scheduled unless current working set can be resident in main memory**
- **Divide runnable processes into two groups:**
  - » **Active: Working set is loaded**
  - » **Inactive: Working set is swapped to disk**
- **Balance set: Sum of working sets of all active processes**
- **Interaction with scheduler**
  - » **If balance set exceeds size of memory, move some process to inactive set**
    - – **Which process???**
  - » **If balance set is less than size of memory, move some process to active set**
    - – **Which process?**
  - » **Any other considerations?**

Working Set Implementation

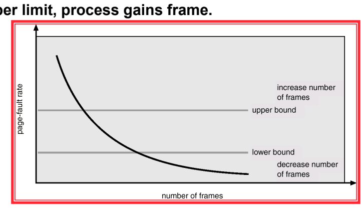
- Leverage `use` bits (as in the clock algorithm)
- OS maintains `idle time` for each page
  - » Amount of CPU received by process since last access to page
  - » Periodically scan all resident pages of a process
    - – If `use` bit is set, clear page's idle time
    - – If `use` bit is clear, add process CPU time (since last scan) to idle time
  - » If idle time < ΔT, page is in working set

Thought Questions

- How should value of ΔT be configured?
  - » What if ΔT is too large?
- How should working set be defined when pages are shared?
  - » Put jobs sharing pages in same balance set
- What processes should compose balance set?
- How much memory is needed for a "balanced system"?
  - » Balanced system: Each resource (e.g., CPU, memory, disk) becomes bottleneck at nearly same time
  - » How much memory is needed to keep the CPU busy?
  - » With working set approach, CPU may be idle even with runnable processes

Page-Fault Frequency Scheme

- Observation: Thrashing has a high page-fault rate
- Idea: Control page fault-rate by controlling # frames that are allocated to a process
  - » Too high page fault rate : process need more frames
  - » Too low : process has too many frames
- Approach: Establish "acceptable" page-fault rate (upper and lower bound)
  - » If actual rate falls below lower limit, process loses frame.
  - » If actual rate exceeds upper limit, process gains frame.



Current Trend: Thoughts?

- VM code is not as critical
  - » Reason #1: Personal vs. time-shared machine
    - – Why does this matter? Clouds?
  - » Reason #2: Memory is more affordable, more memory
- Less hardware support for replacement policies
  - » Software emulation of use and dirty bits
- Larger page sizes
  - » Better TLB coverage
  - » Smaller page tables
  - » Disadvantage: More internal fragmentation
    - – Multiple page sizes