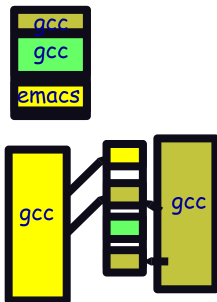
Past: Making physical memory simple

- Physical memory:
  - no protection
  - limited size
  - almost forces contiguous allocation
  - sharing visible to program
  - easy to share data



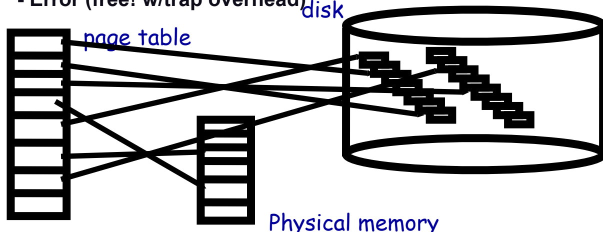
- Virtual memory
  - each program isolated from others
  - transparent:can't tell where running
  - can share code, data
  - non-contiguous allocation
  - Next: some nuances + illusion of infinite memory

Paging

- Readings for this topic: 7th ed. Chapter 9; 6th ed. Chapter 10;
- *Our simple world:*
  - load entire process into memory. Run it. Exit.
- Problems?
  - slow (especially with big process)
  - wasteful of space (process doesn't use all of its memory)
- Solution: partial residency
  - demand paging: only bring in pages actually used
  - paging: only keep frequently used pages in memory
- Mechanism:
  - use virtual memory to map some addresses to physical pages, some to disk

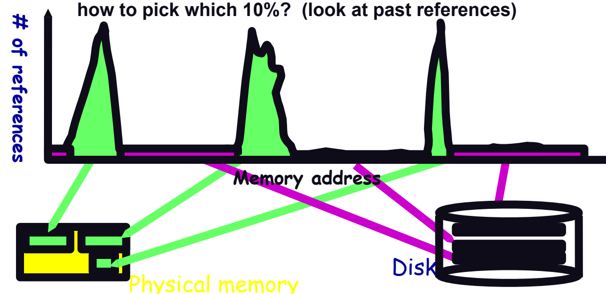
Virtual memory from 50,000 feet

- Virtual address translated to:
  - Physical memory ($20/GB). Very fast, but small
  - Disk ($.20/GB). Very large, but verrrrrry slow (milliseconds vs. nanoseconds)
  - (soon: Flash $4/GB, slow for now)
  - Error (free! w/trap overhead)



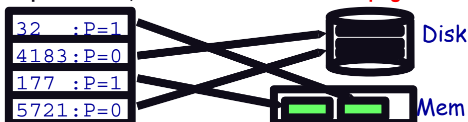
Virtual memory = Our big lie (delayed truth?)

- Want: disk-sized memory that's fast as physical mem
  - 90/10 rule: 10% of memory gets 90% of memory refs
  - so, keep that 10% in real memory, the other 90% on disk
    - how to pick which 10%? (look at past references)



Virtual memory mechanics

- Extend page table entries with extra bit ("present")
  - if page in memory? present = 1, on disk, present = 0
  - translations on entries with present = 1 work as before
  - if present = 0, then translation causes a page fault.



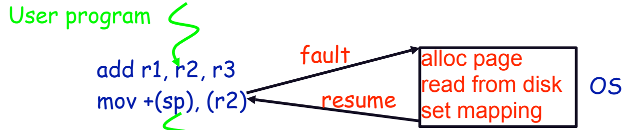
- What happens on page fault?
  - OS finds a free page or evicts one (which one??)
  - issues a disk request to read in data into that page
  - puts process on blocked Q, switches to new process
  - when disk completes: set present = 1, put back on run Q

Virtual memory problems

- Problem 1: how to resume a process after a fault?
  - Need to save state and resume.
  - Process might have been in the middle of an instruction!
- Problem 2: what to fetch?
  - Just needed page or more?
- Problem 3: what to eject?
  - Cache always too small, which page to replace?
  - Want to know future use...

Problem 1: resuming process after a fault

- Fault might have happened in the middle of an inst!



Our key constraint: don't want user process to be aware that page fault happened (just like context switching)

Can we skip the faulting instruction? Uh, no.

Can we restart the instruction from the beginning?

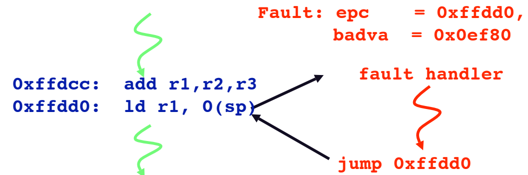
Not if it has partial-side effects.

Can we inspect instruction to figure out what to do?

May be ambiguous where it was.

Solution: a bit of hardware support

- RISC machines are pretty simple:
  - **typically instructions idempotent until references done!**
  - **Thus, only need faulting address and faulting PC.**
- Example: MIPS



- CISC harder:
  - **multiple memory references and side effects**

Problem 2: what to fetch?

- Page selection: when to bring pages into memory
  - **Like all caches: we need to know the future.**
- Doesn't the user know? (Request paging)
  - **Not reliably.**
  - **Though, some OSs do have support for prefetching.**
- Easy load-time hack: demand paging
  - **Load initial page(s). Run. Load others on fault.**

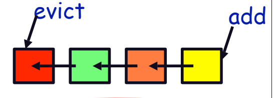
When will startup be slower? Memory less utilized?

Most systems do some sort of variant of this

- Tweak: pre-paging. Get page & its neighbors (why?)

Problem 3: what to eject & when?

- Random: pick any page.
  - **Pro: good for avoiding worst case, simple**
  - **con: good for avoiding best case**
- FIFO: throw out oldest page
  - **fair: all pages get = residency**
  - **dopey: ignores usage.**



- MIN (optimal):
  - **throw out page not used for longest time.**
  - **Impractical, but good yardstick**

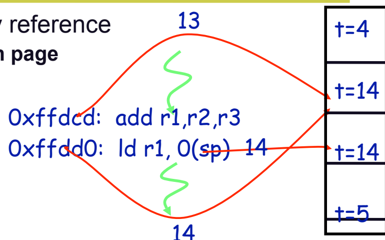
Refs: AGBDCADCABCGABC

evict page

- Least recently used.
  - **throw out page that hasn't been used in the longest time.**
  - **Past = future? LRU = MIN.**

Implementing Perfect LRU

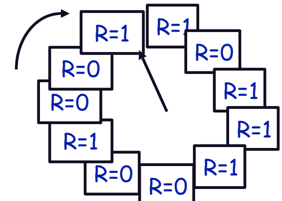
- On every memory reference
  - **Time stamp each page**
- At eviction time:
  - **Scan for oldest**



- Problems:
  - **Large page lists**
  - **No hardware support for time stamps**
- "Sort of" LRU
  - **Do something simple & fast that finds an old page**
  - **LRU an approximation anyway, a little more won't hurt…**

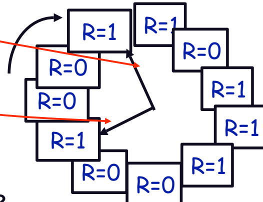
LRU in the real world: the clock algorithm

- Each page has reference bit
  - **Hardware sets on use, OS periodically clears**
  - **Pages with bit set used more recently than without.**
- Algorithm: FIFO + skip referenced pages
  - **Keep pages in a circular FIFO list**
  - **Scan: page's ref bit = 1, set to 0 & skip, otherwise evict.**
- Good? Bad?:
  - **Hand sweeping slow?**
  - **Hand sweeping fast?**



Problem: what happens as memory gets big?

- Sol'n: add another clock hand
  - **Leading edge clears ref bits**
  - **Trailing edge is "C" pages back: evicts pages w/ 0 ref bit**



- Implications:
  - **Angle too small?**
  - **Angle too large?**

BSD Unix: Clock algorithm in Action!

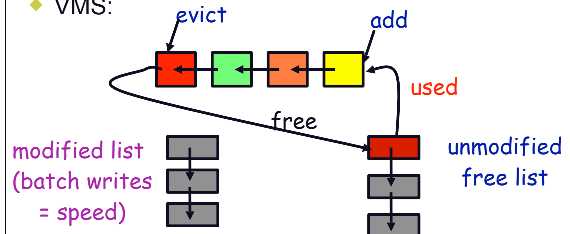
- used vmstat on Solaris to see
  - **elaine6: vmstat -s # -s: pages scanned by clock/second**
    - **2*92853 pages examined by the clock daemon**
    - **6 revolutions of the clock hand**
    - **127878 pages freed by clock daemon**
  - **leland: vmstat -s # large, shared machine**
    - **14157550 pages examined by clock daemon**
    - **13065972 pages freed by clock daemon**
    - **110 revolutions of clock hand # since boot!**
    - **3482069 forks**
    - **13057795 pages freed by clock daemon**
  - **csl: vmstat -s # smaller machine**
    - **15086 revolutions of the clock hand # buy more mem!**
    - **672474 forks**

The clock algorithm improved

- Problem: crude & overly sensitive to sweeping interval
  - **Infrequent? All pages look used.**
  - **Frequent? Lose too much usage information**
  - **Simple changes = more accurate & robust w/ ~same work**
- Clock: 1 bit per page
  - **When page used: set use bit**
  - **Sweep: clear use bit**
  - **Select page? FIFO + skip if use bit set**
- Clock': n bits per page
  - **When page used: set use bit**
  - **Sweep: use_count = (use_bit << n-1) | (use_count >> 2)**
    - **(why shift?)**
  - **Select page? take lowest use count**

A different take: page buffering

- Cute simple trick (VMS, Mach, Windows NT..Vista):
  - **Keep spare of free pages; recycle in FIFO order**
  - **but record what free page corresponds to: if used before overwritten put back in place.**
- VMS:



Global or local?

- So far, we've implicitly assumed memory comes from a single global pool - "Global replacement"
  - **When process P faults and needs a page, take oldest page on entire system**
  - **Good: shared caches are adaptable. Example if P1 needs 20% of memory and P2 70%, then they will be happy.**

  - **Bad: too adaptable. No protection from resource hogs**
    - **What happens to P1 if P2 sequentially reads array which is about the size of our total memory?**

Per-process and per-user page replacement

- Per-process (per-user same)
  - **Each process has a separate pool of pages**
  - **A page fault in one process can only replace one of this process's frames**
  - **Isolates process and therefore relieves interference from other processes**

pool barrier
P1
P2

  - **But, isolates process and therefore prevents process from using other's (comparatively) idle resources**
  - **Efficient memory usage requires a mechanism for (slowly) changing the allocations to each pool**
  - **Qs: What is "slowly"? How big a pool? When to migrate?**

Why does VM caching look so different?

- Recall: formula for average access time w/cache:

  p = % of accesses that hit in cache
  access time = p * (cache access time)
  + (1-p) * (real access time + cache miss overhead)

- TLB and memory cache need access times ~ that of instruction

  Not a whole lot of time to play around.

- VM caching measured closer to that of a disk access.

  Miss cost so expensive, easily hide high associativity cost, and overhead of sophisticated replacement algorithms