CSE 451: Operating Systems
Autumn 2001

Lecture 11
Demand Paging and
Page Replacement

Demand Paging

- We've hinted that pages can be moved between memory and disk
  - this process is called demand paging
    - is different than swapping (entire process moved, not page)
  - OS uses main memory as a (page) cache of all of the data allocated by processes in the system
    - initially, pages are allocated from physical memory frames
    - when physical memory fills up, allocating a page in requires some other page to be evicted from its physical memory frame
  - evicted pages go to disk (only need to write if they are dirty)
    - to a swap file
    - movement of pages between memory / disk is done by the OS
    - is transparent to the application
      - except for performance…

Page Faults

- What happens to a process that references a VA in a page that has been evicted?
  - when the page was evicted, the OS sets the PTE as invalid and stores (in PTE) the location of the page in the swap file
  - when a process accesses the page, the invalid PTE will cause an exception (page fault) to be thrown
  - the OS will run the page fault handler in response
    - handler uses invalid PTE to locate page in swap file
    - handler reads page into a physical frame, updates PTE to point to it and to be valid
    - handler restarts the faulted process
- But: where does the page that's read in go?
  - have to evict something else (page replacement algorithm)
    - OS typically tries to keep a pool of free pages around so that allocations don't inevitably cause evictions

What do you do to pages?

- If the page is dirty, you have to write it out to disk.
  - record the disk block number for the page in the PTE.
- If the page is clean, you don't have to do anything.
  - just overwrite the page with new data
  - make sure you know where the old copy of the page came from
- Want to avoid THRASHING
  - When a paging algorithm breaks down
  - Most of the OS time spent in ferrying pages to and from disk
  - no time spent doing useful work.
  - the system is OVERCOMMITTED
    - no idea what pages should be resident in order to run effectively
  - Solutions include:
    - SWAP
    - Buy more memory

Why does demand paging work?

- Locality!
  - temporal locality
    - locations referenced recently tend to be referenced again soon
  - spatial locality
    - locations near recently references locations are likely to be referenced soon (think about why)
- Locality means paging can be infrequent
  - once you've paged something in, it will be used many times
  - on average, you use things that are paged in
  - but, this depends on many things:
    - degree of locality in application
    - page replacement policy and application reference pattern
    - amount of physical memory and application footprint

Why is this “demand” paging?

- Think about when a process first starts up:
  - it has a brand new page table, with all PTE valid bits ‘false’
  - no pages are yet mapped to physical memory
  - when process starts executing:
    - instructions immediately fault on both code and data pages
    - faults stop when all necessary code/data pages are in memory
    - only the code/data that is needed (demanded!) by process needs to be loaded
    - what is needed changes over time, of course…

Finding the Best Page

- A good property
  - if you put more memory on the machine, then your page fault rate will go down.
  - Increasing the size of the resource pool helps everyone.
- The best page to toss out is the one you'll never need again
  - that way, no faults.
- Never is a long time, so picking the one closest to "never" is the next best thing.
  - Replacing the page that won't be used for the longest period of time absolutely minimizes the number of page faults.
  - Example:
    - **B C B A E B D E C B E B**
    - **Three page frames**
    - **what page to toss on each fault?**

Evicting the best page

- The goal of the page replacement algorithm:
  - reduce fault rate by selecting best victim page to remove
  - the best page to evict is one that will never be touched again
    - as process will never again fault on it
  - "never" is a long time
    - Belady's proof: evicting the page that won't be used for the longest period of time minimizes page fault rate
- Rest of this lecture:
  - survey a bunch of replacement algorithms

Optimal Algorithm

- The optimal algorithm, called Belady's algorithm, has the lowest fault rate for any reference string.
- Basic idea: replace the page that will not be used for the longest time in the future.
- Basic problem: hard to know the future
- Basic use: gives us an idea of how well any implementable algorithm is doing relative to the best possible algorithm.
  - compare the fault rate of any proposed algorithm to Optimal
  - if Optimal does not do much better, then your proposed algorithm is pretty good.
  - If your proposed algorithm doesn't do much better than Random, go home.

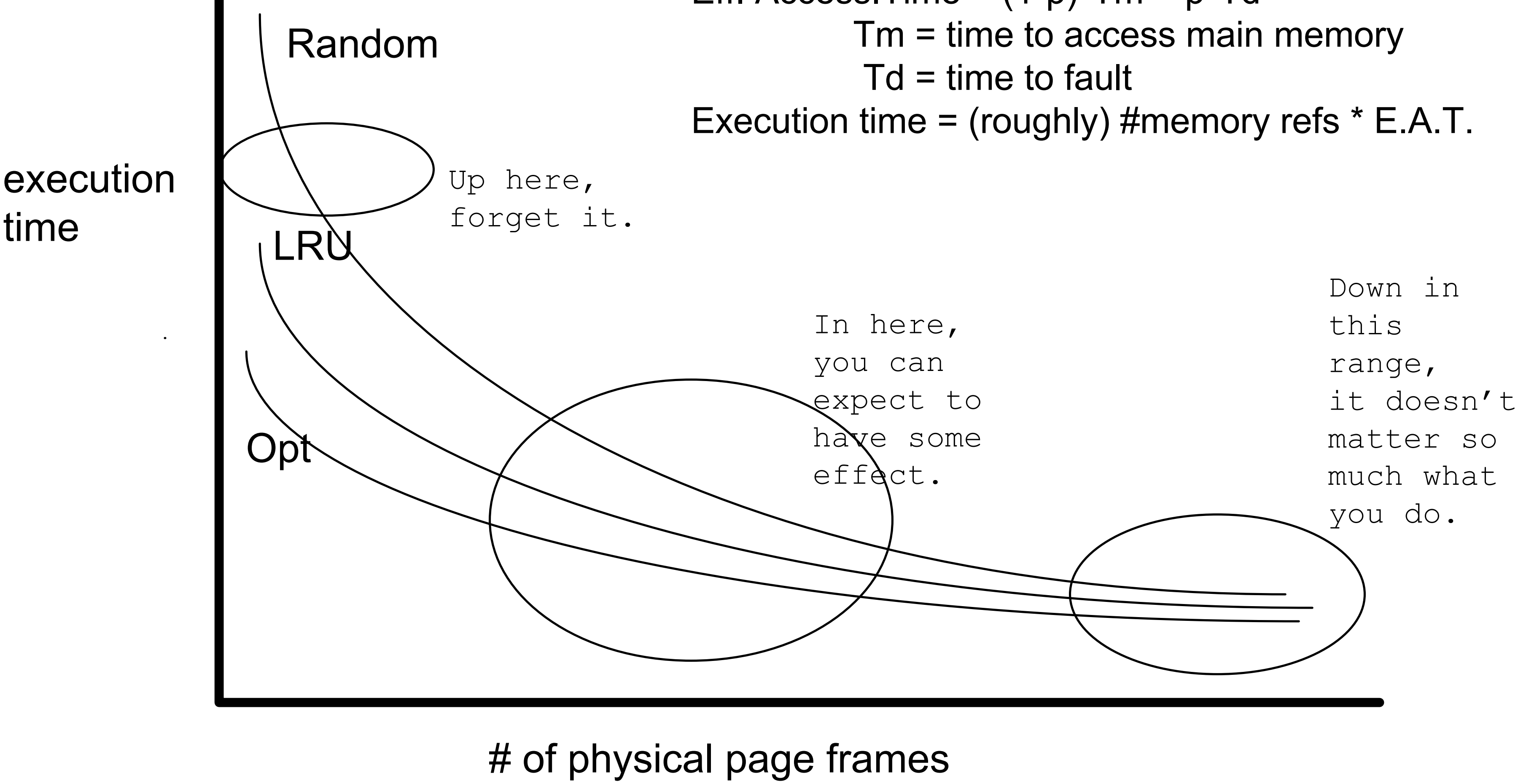
Evaluating Replacement Policies

Eff. Access.Time = (1-p)*Tm + p*Td

Tm = time to access main memory

Td = time to fault

Execution time = (roughly) #memory refs * E.A.T.



#1: Belady's Algorithm

- Provably optimal lowest fault rate
  - pick the page that won't be used for longest time in future
  - problem: impossible to predict future
- Why is Belady's algorithm useful?
  - as a yardstick to compare other algorithms to optimal
    - if Belady's isn't much better than yours, yours is pretty good
- Is there a lower bound?
  - unfortunately, lower bound depends on workload
    - but, random replacement is pretty bad

#2: FIFO

- FIFO is obvious, and simple to implement
  - when you page in something, put in on tail of list
  - on eviction, throw away page on head of list
- Why might this be good?
  - maybe the one brought in longest ago is not being used
- Why might this be bad?
  - then again, maybe it is being used
  - have absolutely no information either way
- FIFO suffers from Belady's Anomaly
  - fault rate might increase when algorithm is given more physical memory
    - a very bad property

An Example of Optimal and FIFO in Action

```
Reference stream is A B C A B D A D B C

OPTIMAL
        A  B  C  A  B  D  A  D  B  C  B

        5 Faults
```

toss C

toss A or D

A
B
C
D
A
B
C

```
FIFO
        A  B  C  A  B  D  A  D  B  C  B

        7 Faults
```

toss A

toss ?

#3: Least Recently Used (LRU)

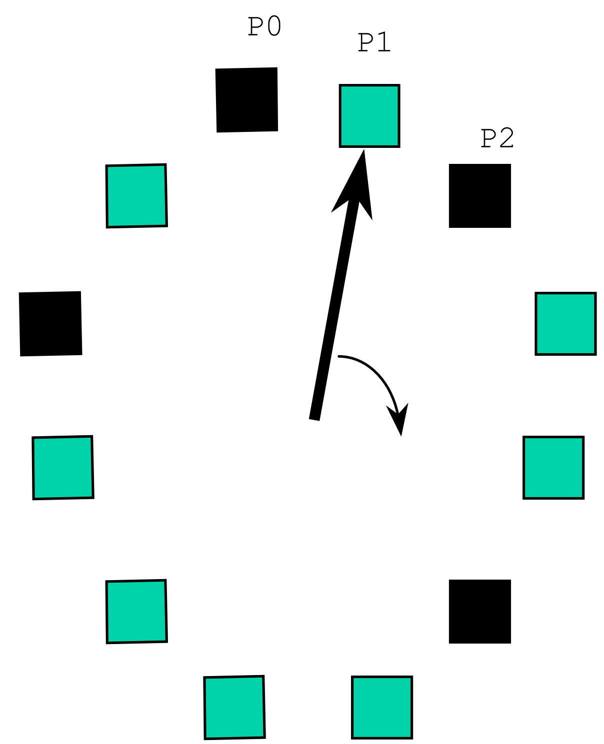
- LRU uses reference information to make a more informed replacement decision
  - idea: past experience gives us a guess of future behavior
  - on replacement, evict the page that hasn't been used for the longest amount of time
    - LRU looks at the past, Belady's wants to look at future
  - when does LRU do well?
    - when does it suck?
- Implementation
  - to be perfect, must grab a timestamp on every memory reference and put it in the PTE (way too $$)
  - so, we need an approximation…

Approximating LRU

- Many approximations, all use the PTE reference bit
  - keep a counter for each page
  - at some regular interval, for each page, do:
    - if ref bit = 0, increment the counter (hasn't been used)
    - if ref bit = 1, zero the counter (has been used)
    - regardless, zero ref bit
  - the counter will contain the # of intervals since the last reference to the page
    - page with largest counter is least recently used
- Some architectures don't have PTE reference bits
  - can simulate reference bit using the valid bit to induce faults
    - hack, hack, hack

#4: LRU Clock

- AKA Not Recently Used (NRU) or Second Chance
  - replace page that is "old enough"
  - arrange all physical page frames in a big circle (clock)
    - just a circular linked list
  - a "clock hand" is used to select a good LRU candidate
    - sweep through the pages in circular order like a clock
    - if ref bit is off, it hasn't been used recently, we have a victim
      - so, what is minimum "age" if ref bit is off?
    - if the ref bit is on, turn it off and go to next page
  - arm moves quickly when pages are needed
  - low overhead if have plenty of memory
  - if memory is large, "accuracy" of information degrades
    - add more hands to fix

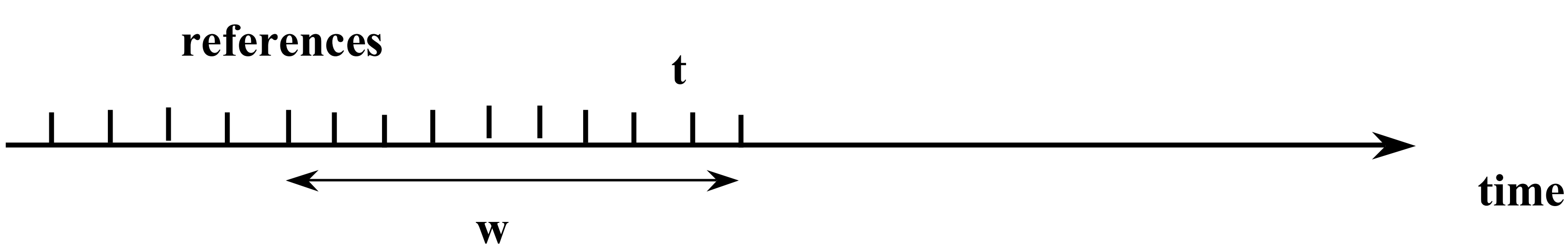


Another Problem: allocation of frames

- In a multiprogramming system, we need a way to allocate physical memory to competing processes
  - what if a victim page belongs to another process?
  - family of replacement algorithms that takes this into account
- Fixed space algorithms
  - each process is given a limit of pages it can use
  - when it reaches its limit, it replaces from its own pages
  - local replacement: some process may do well, others suffer
- Variable space algorithms
  - processes' set of pages grows and shrinks dynamically
  - global replacement: one process can ruin it for the rest
    - linux uses global replacement

Important concept: working set model

- A working set of a process is used to model the dynamic locality of its memory usage
  - i.e., working set = set of pages process currently “needs”
  - formally defined by Peter Denning in the 1960’s
- Definition:
  - WS(t,w) = {pages P such that P was referenced in the time interval (t, t-w)}
    - t – time, w – working set window (measured in page refs)
    - a page is in the working set (WS) only if it was referenced in the last w references

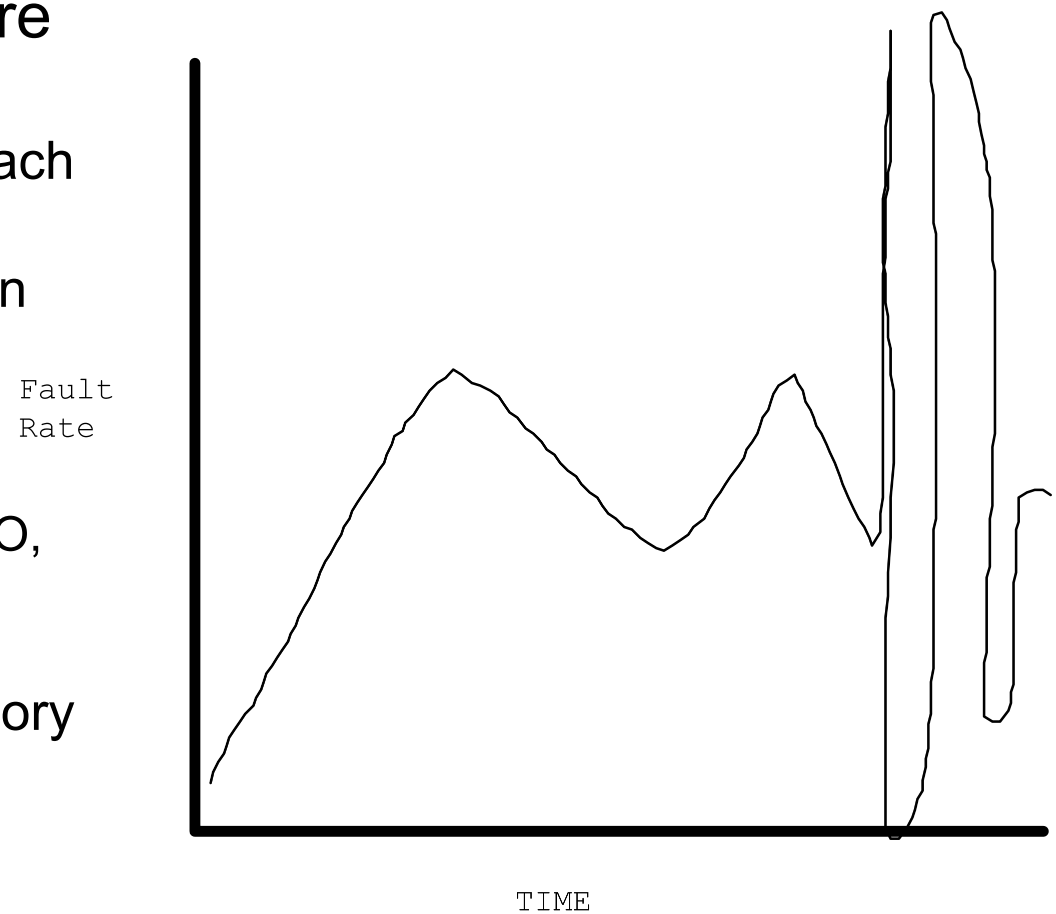


#5: Working Set Size

- The working set size changes with program locality
  - during periods of poor locality, more pages are referenced
  - within that period of time, the working set size is larger
- Intuitively, working set must be in memory, otherwise you'll experience heavy faulting (thrashing)
  - when people ask "How much memory does Netscape need?", really they are asking "what is Netscape's average (or worst case) working set size?"
- Hypothetical algorithm:
  - associate parameter "w" with each process
  - only allow a process to start if it's "w", when added to all other processes, still fits in memory
    - use a local replacement algorithm within each process
- But, we have two problems:
    - **how do we select w?**
    - **how do we know when the working set changes?**
- So, working set is not used in practice.

#6: Page Fault Frequency (PFF)

- PFF is a variable-space algorithm that uses a more ad-hoc approach
  - monitor the fault rate for each process
  - if fault rate is above a given threshold, give it more memory
    - so that it faults less
    - doesn't always work (FIFO, Belady's anomaly)
  - if the fault rate is below threshold, take away memory
    - should fault more
    - again, not always



Thrashing

- What the OS does if page replacement algo's fail
  - happens if most of the time is spent by an OS paging data back and forth from disk
    - no time is spent doing useful work
    - the system is overcommitted
    - no idea which pages should be in memory to reduced faults
    - could be that there just isn't enough physical memory for all processes
  - solutions?
- Yields some insight into systems research[ers]
  - if system has too much memory
    - page replacement algorithm doesn't matter (overprovisioning)
  - if system has too little memory
    - page replacement algorithm doesn't matter (overcommitted)
  - problem is only interesting on the border between overprovisioned and overcommitted
    - many research papers live here, but not many real systems do…

Summary

- demand paging
  - start with no physical pages mapped, load them in on demand
- page replacement algorithms
  - #1: Belady's – optimal, but unrealizable
  - #2: Fifo – replace page loaded furthest in past
  - #3: LRU – replace page referenced furthest in past
    - approximate using PTE reference bit
  - #4: LRU Clock – replace page that is "old enough"
  - #5: working set – keep set of pages in memory that induces the minimal fault rate
  - #6: page fault frequency – grow/shrink page set as a function of fault rate
- local vs. global replacement
  - should processes be allowed to evict each other's pages?