Stack Algorithms, Working Set Model, Copy-on-write, VM Review

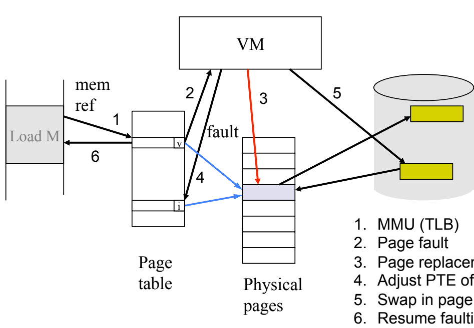
ECE595
Oct 30

Y. Charlie Hu

[lec16] Virtual Memory

- Definition: *Virtual memory* permits a process to run with only some of its virtual address space loaded into physical memory
- VM is typically implemented by demand paging
  - Virtual address space translated to either
    - Physical memory (small, fast) or
    - Disk (backing store), large but slow
- Objective:
  - To produce the illusion of memory as big as necessary

[lec16] Page Fault Handling in demand paging



1. MMU (TLB)
2. Page fault
3. Page replacement
4. Adjust PTE of victim pg
5. Swap in page from swap
6. Resume faulting intr

[lec18] Page replacement algorithms: Summary

- Optimal
- FIFO
- Random
- FIFO with 2nd chance
- Clock: a simple FIFO with 2nd chance
- Enhanced FIFO with 2nd chance
- Approximate LRU

"Deep thinking"

- For a fixed replacement algorithm, more page frames → fewer page faults?

More Page Frames → Fewer Page Faults?

- Consider the following reference string with 4 physical pages
  - FIFO replacement
  - **1, 2, 3, 4**, *1, 2*, **5, 1, 2, 3, 4, 5**
  - How many page faults?
- Consider the same reference string with 3 physical pages
  - FIFO replacement
  - How many page faults?
- This is called *Belady's anomaly*

Stack algorithms

- Definition: a page replacement algorithm in which it can be shown that the set of pages that would be in memory for *n* physical pages is *always a subset* of the set of pages that would be in memory for *n+1* physical pages
- Implication: hit rate of stack algorithms never decreases when number of physical pages grows
- Examples: OPT? LRU? FIFO? LFU?

OPT is a stack algorithm

- Proof by induction:
  - Given
    - A mem X of N physical pages
    - A mem Y of N+1 physical pages
    - A sequence of virtual page accesses
  - The claim is true after i accesses,
  - On the (i+1)th access of x, outcome on two mems are
    - (h,h) -- trivial
    - (m,h) -- (2, 4, 3), (2, 4, 3, 7) → (2, 4, 7), (2, 3, 4, 7)
    - (m,m) – (2, 4, 3), (2, 4, 3, 7)
      - Case 1: 7 is used furthest
      - Case: 2/4/3 is used furthest
  - Show after replacement, claim is still true

OPT is a stack algorithm

- Is there a one-sentence argument?

The BIG picture

- We've talked about single evictions
- Most computers are multiprogrammed
  - Single eviction decision still needed
  - New concern – processes compete for resources
  - How to be "fair enough" and achieve good overall throughput

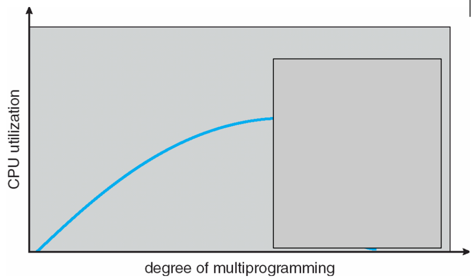
Possible replacement strategies

- Global replacement:
  - All pages from all processes are lumped into a single replacement pool
  - Most flexibility, least "pig protection"
- Local replacement
  - Per-process replacement:
    - Each process has a separate pool of pages
  - Per-user replacement:
    - Lump all processes for a given user into a single pool
- In local replacement, must have a mechanism for (slowly) changing the allocations to each pool

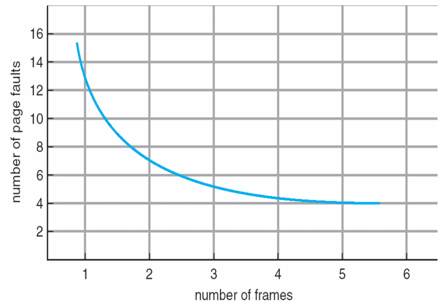
Improving CPU utilization in multiprogramming

- In multiprogramming, when OS sees the CPU utilization is low,
  - It thinks most processes are waiting for I/O
  - it needs to increase the degree of multiprogramming (actual behavior of early paging systems)
  - It adds/loads another process to the system
    - Assume I/O capacity is large, every job spends 50% of time performing I/O, how many such jobs are needed to keep CPU 100% utilized?

Towards Improving CPU utilization



Page faults versus number of physical pages for a process



When there are not enough page frames

- Suppose many processes are making frequent references to 50 pages, memory has 49
- Assuming LRU
  - Each time one page is brought in, another page, whose content will soon be referenced, is thrown out
- What is the average memory access time?
- The system is spending most of its time paging!
- The progress of programs makes it look like "*memory access is as slow as disk*", rather than "*disk being as fast as memory*"

Thrashing

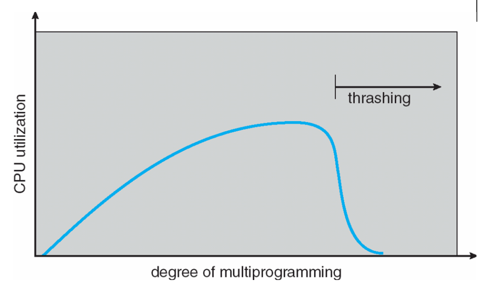
- **Thrashing** ≡ a process is busy swapping pages in and out

Thrashing can lead to vicious cycle

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - OS thinks that it needs to increase the degree of multiprogramming (actual behavior of early paging systems)
  - another process added to the system
  - page fault rate goes even higher

Vicious Cycle

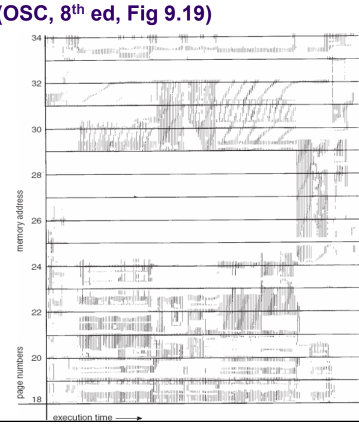
Thrashing (Cont.)



Demand paging and thrashing

- Why does demand paging work?
  - Data reference exhibits locality
- Why does thrashing occur?
  - Σ size of locality > total memory size

Locality in a memory-reference pattern (OSC, 8th ed, Fig 9.19)

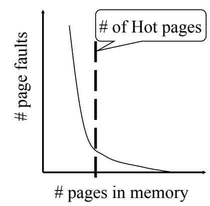


Intuitively, what to do about thrashing?

- If a single process's locality too large for memory, what can OS do?
- If the problem arises from the sum of several processes?
  - Figure out how much memory each process needs – "locality"
  - What can we do?
    - Can limit effects of thrashing using local replacement
    - Or, bring a process' working set before running it
    - Or, wait till there is enough mem for a process's need

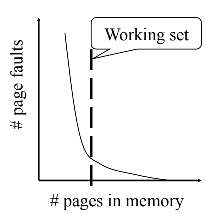
Key observation

- Locality in memory references
  - Spatial and temporal
- Want to keep a set of pages in memory that would avoid a lot of page faults
  - "Hot" pages
- Can we formalize it?



Working Set Model – by Peter Denning (Purdue CS head, 1979-83)

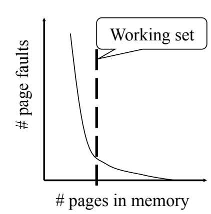
- An informal definition:
  - Working set: The collection of pages that a process is working with, and which must thus be resident if the process is to avoid thrashing



- But how to turn the concept/theory into practical solutions?
  1. Capture the working set
  2. Influence the scheduler or replacement algo

Working Set Model – by Peter Denning (Purdue CS head, 79-83)

- Usage idea: use recent needs of a process to predict its future needs
  - Choose δ, the WS parameter
  - At any given time, all pages referenced by a process in its last δ seconds comprise its working set
  - Don't execute a process unless there is enough mem to fit its working set
- Needs a companion replacement algo



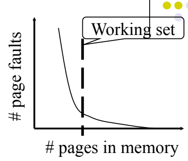
Working Set replacement algorithm

- Main idea
  - *Take advantage of reference bits*
  - *Variation of FIFO with 2nd chance*
- An algorithm (assume reference bit)
  - On a page fault, scan through all pages of the process
  - If the reference bit is 1, clear the bit, record the current time for the page
  - If the reference bit is 0, check the "last use time"
    - If the page has not been used within δ, replace the page
    - Otherwise, go to the next page

Working Set Clock Algorithm 2
(assume reference bit + modified bit)

- Upon page fault, follow the clock hand
- If the reference bit is 1, set reference bit to 0, set the current time for the page and go to the next
- If the reference bit is 0, check "last use time"
  - If page used within δ, go to the next
  - If page not used within δ and modify bit is 1
    - Schedule the page for page out (then reset modify bit) and go to the next
  - If page not used within δ and modified bit is 0
    - Replace this page

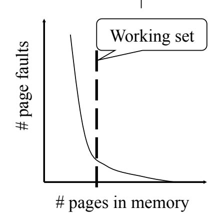
Challenges with WS algorithm implementation



- What should δ be?
  - What if it is too large?
  - What if it is too small?
- How many jobs need to be scheduled in order to keep CPU busy?
  - Too few → cannot keep CPU busy if all doing I/O
  - Too many → their WS may exceed memory

More Challenges with Capturing Working Set

- Working set isn't static
- There often isn't a single "working set"
  - e.g., Multiple plateaus in previous curve (L1 $, L2 $, etc)
  - Program coding style affects working set

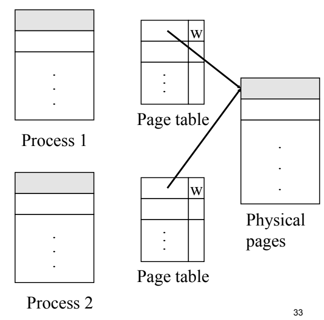


- Working set is often hard to gauge
  - What's the working set of an interactive program?
  - How to calculate WS if pages are shared?

Break

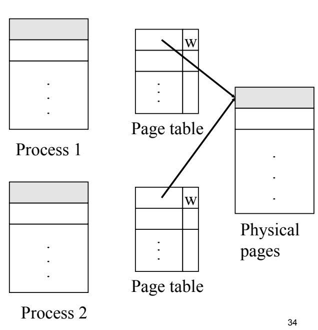
Shared Memory

- How do two processes share memory under paging?



With Shared Memory

- How to destroy a virtual address space?
  - Reference count
- How to swap out/in?
  - Link all PTEs
  - Operation on all entries
- How to pin/unpin?
  - Link all PTEs
  - Reference count



[lec4] C program Forking a new Process

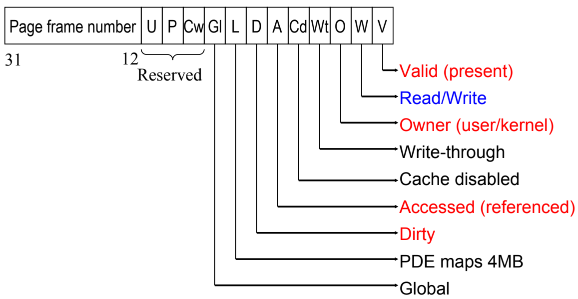
```
#include <stdio.h>

void main()
{
  int pid;       int was = 3;
  pid = fork(); /* fork another process */

  if (pid == 0) { /* child process */
     sleep(2); printf(“was = %d”, was);
     execlp(“/bin/ls”, “ls”, NULL);}
  else { /* pid > 0; parent process */
     was = 4;
     printf(“child process id %d was=%d ”, pid, was);
     wait(NULL);    exit(0);
  }
}
```

- How to efficiently implement fork()?

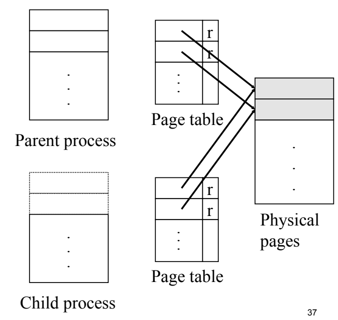
x86 Page Table Entry



Copy-On-Write

Used in Win2k,
Linux Solaris2
in duplicating processes

- Child's virtual address uses the same page mapping as parent's
- Make all pages read-only
- On a read, nothing happens
- On a write, generates an access fault
  - map to a new page frame
  - copy the page over
  - restart the instruction



Virtual Memory Review (1/3)

- Page fault handling (mechanism)
- Paging algorithms (policy)
  - Optimal
  - FIFO
  - FIFO with 2nd chance
  - Clock: a simple FIFO with 2nd chance
  - LRU
  - Approximate LRU
  - NFU

Virtual Memory Review (2/3)

- Important questions
  - What is the use of optimal algo?
  - If future is unknown, what make us think there is a chance for doing a good job?
  - Without addi. hardware support, the best we can do?
  - What is the minimal hardware support under which we can do a decent job?
  - Why is it difficult to implement exact LRU? Exact anythg
  - For a fixed replacement algo, more page frames → less page faults?
  - What are stack algorithms?
  - How can we move page-out out of critical path?

Virtual Memory Review (3/3)

- Per-process vs. global page replacement
- Stack algorithms
  - OPT, LRU, LFU, FIFO?
- Thrashing
  - What causes thrashing?
  - What to do about thrashing?
  - What is working set?