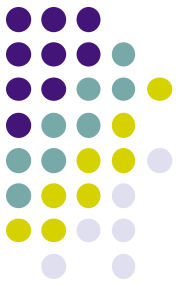


Virtual Memory Review (1/3)

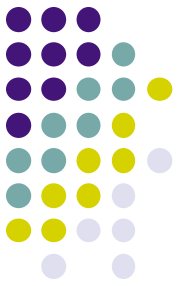
- Page fault handling (mechanism)
- Paging algorithms (policies)
 - Optimal
 - FIFO
 - FIFO with 2nd chance
 - Clock: a simple FIFO with 2nd chance
 - LRU
 - Approximate LRU
 - NFU



Virtual Memory Review (2/3)

- Important questions

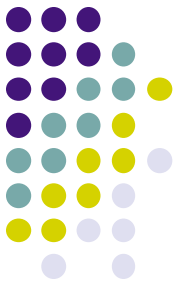
- What is the use of optimal algo?
- If future is unknown, what makes us think there is a chance for doing a good job?
- Without additional hardware support, the best we can do?
- What is the minimal hardware support under which we can do a decent job?
- Why is it difficult to implement exact LRU? (exact anything)
- For a fixed replacement algorithm, more page frames → less page faults?
- How can we move page-out out of critical path?



Virtual Memory Review (3/3)

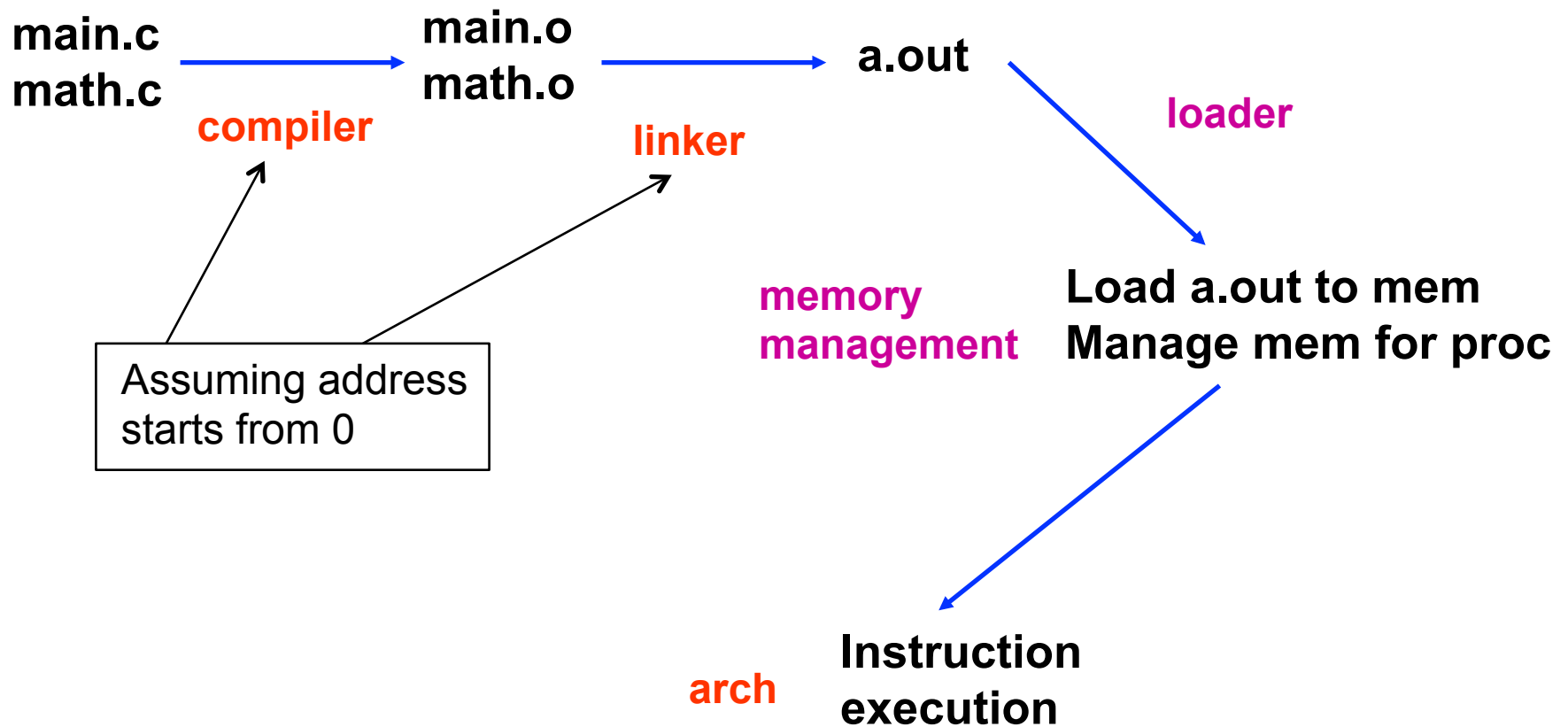
- Per-process vs. global page replacement
- Thrashing
- What causes thrashing?
- What to do about thrashing?
- What is working set?
- Working set replacement algorithms
- Memory sharing and copy-on-write

Page replacement algorithms: Summary



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

[lec13] The big picture – connecting the dots



Virtual Memory, Demand Paging



[lec15] Sharing main memory

- Simple multiprogramming – 4 drawbacks
 - Lack of protection
 - Cannot relocate dynamically
 - dynamic memory relocation: base&bound
 - Single segment per process
 - dynamic memory relocation: segmentation, paging
- Entire address space needs to fit in mem
 - More need for swapping
 - Need to swap whole, very expensive!

[lec15] 3. Dynamic memory relocation



- Instead of changing the address of a program before it's loaded, change the address dynamically *during every reference*
 - Under dynamic relocation, each program-generated address (called a *logical address* or *virtual address*) is translated in hardware to a *physical* or *real address* at runtime



The last drawback

- So far we've separated the process's view of memory from the OS's view using a mapping mechanism
 - Each sees a different organization
 - Allows OS to shuffle processes around
 - Simplifies memory sharing
 - *What is the essence of the mechanism that enables this?*
 - But, a user process had to be completely loaded into memory before it could run
- Wasteful since a process only needs a small amount of its total memory at any time (*reference locality!*)

Virtual Memory



- Definition: *Virtual memory* permits a process to run with only some of its virtual address space loaded into physical memory
- Key idea: Virtual address space translated to either
 - Physical memory (small, fast) or
 - Disk (backing store), large but slow
- Deep thinking – what made above possible?
- Objective:
 - To produce the illusion of memory as big as necessary



Virtual Memory

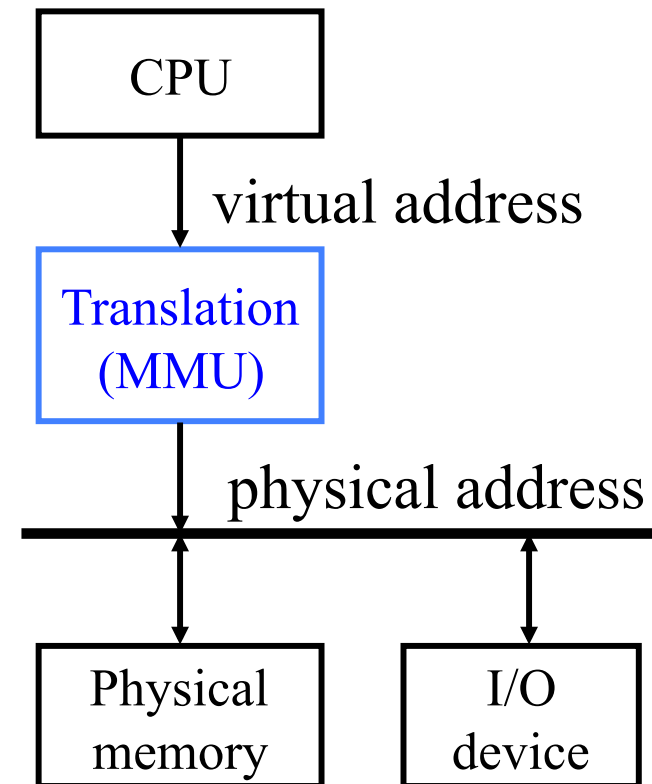
- “To produce the illusion of memory as big as necessary”
 - Without suffering a huge slowdown of execution
 - What makes this possible?
 - *Principle of locality*
 - Knuth’s estimate of 90% of the time in 10% of the code
 - There is also significant locality in data references

Demand Paging **(paging with swapping)**



- If not all of a program is loaded when running, what happens when referencing a byte not loaded yet?

- How to **detect** this?
 - In software?

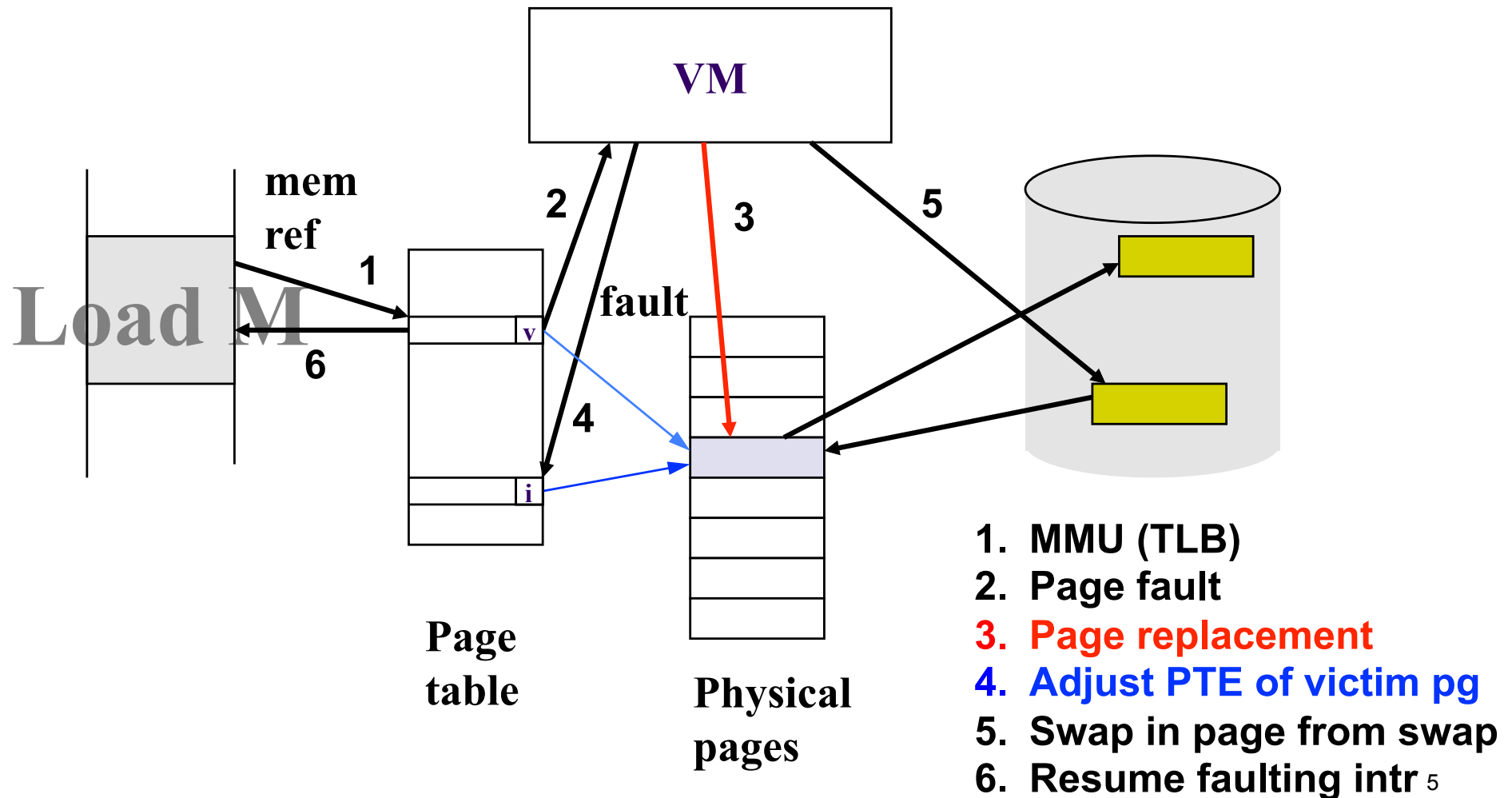
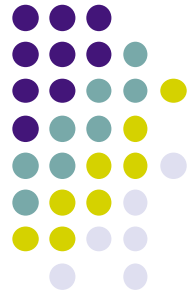


Demand Paging (paging with swapping)



- If not all of a program is loaded when running, what happens when referencing a byte not loaded yet?
- Hardware/software cooperate to make things work
 - Extend PTEs with an **extra bit** “present”
 - Any page not in main memory right now has the “present” bit cleared in its PTE
 - If “present” isn’t set, a reference to the page results in a trap by the paging hardware, called *page fault*
 - What needs to happen when page fault occurs?

[lec16] Page Fault Handling in demand paging

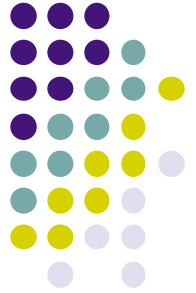




Page fault handling (cont)

- On a page fault
 - Find an unused phy. page or a used phy. page (how?)
 - If the phy. page is used
 - If it has been modified (how to know?), write it to disk
 - Invalidate its current PTE and TLB entry (how?)
 - Load the new page from disk
 - Update the faulting PTE and its TLB entry
 - Restart the faulting instruction
- Supporting data structure
 - For speed: A list of unused physical pages (more later)
 - Data structure to map a phy. page to its pid and virtual address
 - Sounds familiar?

[lec16] The Policy Issue: Page selection and replacement



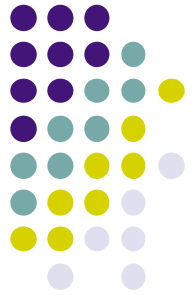
- Once the hardware has provided basic capabilities for VM, the OS must make two kinds of decisions
 - **Page selection:** *when* to bring pages into memory
 - **Page replacement:** *which* page(s) should be thrown out
 - Try to kick out the least useful page



Page selection (when)

- Prepaging
 - Allocate physical page and (if from swap) bring a page into memory before it is referenced
 - Hard to do without a “prophet”, may unnecessarily bring pages
- Request paging
 - Let user say which pages are needed when
 - Users don't always know best
 - And aren't always impartial
- Demand paging
 - Start up process with no pages loaded
 - Load a page when a page fault occurs, i.e., wait till it MUST be in memory
 - Almost all paging systems are demand paging

[lec 17] Page replacement problem



Definition:

- Expect to run with all physical pages in use
 - Every “page-in” requires an eviction to swap space
 - How to select the victim page?
-
- Performance goal:
 - Give a sequence of memory accesses, minimize the # of page faults
 - given a sequence of virtual page references, minimize the # of page faults
-
- Intuition: kick out the page that is least useful
 - Challenge: how do you determine “least useful”?
 - Even if you know future?

The BIG picture: Running at Memory Capacity



- Expect to run with all phy. pages in use
- Every “page-in” requires an eviction
- Goal of page replacement
 - Maximize hit rate → kick out the page that's least useful
- Challenge: how do we determine utility?
 - Kick out pages that aren't likely to be used again
- Page replacement is a difficult policy problem

Definitions

(or, jargons asked during interviews)



- **Pressure** – the demand for some resource (often used when demand exceeds supply)
ex: the system experienced memory pressure
- **Eviction** – throwing something out
ex: cache lines and memory pages got evicted
- **Pollution** – bringing in useless pages/lines
ex: this strategy causes high cache pollution



More definitions

- **Thrashing** – extremely high rate of paging, usually induced by other decisions
- **Locality** – re-use – it makes the world go rounds!
- **Temporal Locality** – re-use in time
- **Spatial Locality** – re-use of close by locations

Refinement by adding extra hardware support



- **Reference bit**
 - A hardware bit that is set whenever the page is referenced (read or written)
- **Modified bit (dirty bit)**
 - A hardware bit that is set whenever the page is written into

Enhanced FIFO with 2nd-Chance Algorithm (used in Macintosh VM)



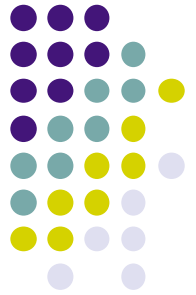
- Same as the basic FIFO with 2nd chance, except that it considers both (reference bit, modified bit)
 - (0,0): neither recently used nor modified (good)
 - (0,1): not recently used but dirty (not as good)
 - (1,0): recently used but clean (not good)
 - (1,1): recently used and dirty (bad)
 - When giving second chance, only clear reference bit
- Pros
 - Avoid write back
- Cons
 - More complicated, worse case scans multiple rounds



Enhanced FIFO with 2nd-Chance Algorithm – implementation



- On page fault, follow hand to inspect pages:
 - Round 1:
 - If bits are (0,0), take it
 - if bits are (0,1), record 1st instance
 - Clear ref bit for (1,0) and (1,1), if (0,1) not found yet
 - At end of round 1, if (0,1) was found, take it
 - If round 1 does not succeed, try 1 more round

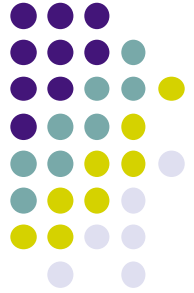


Implementing LRU: hardware

- A counter for each page
- Every time page is referenced, save system clock into the counter of the page
- Page replacement: scan through pages to find the one with the oldest clock
- Problem: have to search all pages/counters!

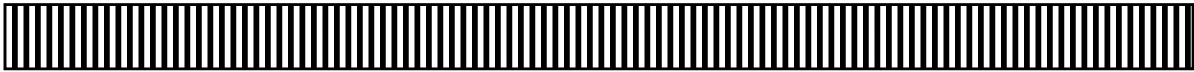
Page Replacement



Approximate LRU



Most recently used

Least recently used

Exact LRU  N categories
pages in order of last reference

Crude LRU   2 categories (roughly)
pages referenced since the last page fault pages not referenced since the last page fault

8-bit count

0	1	2	3
---	---	---	---

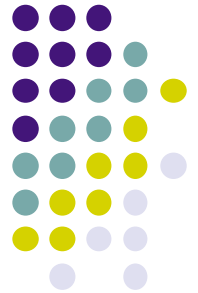
 . . .

254	255
-----	-----

 256 categories

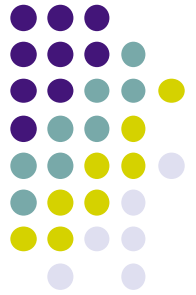
Keep 8-bit counter for each page in a table in memory

Approximate LRU



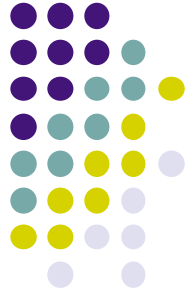
Interval 1	Interval 2	Interval 3	Interval 4	Interval 5
00000000	00000000	10000000	01000000	10100000
00000000	10000000	01000000	10100000	01010000
10000000	11000000	11100000	01110000	00111000
00000000	00000000	00000000	10000000	01000000

- Algorithm
 - At regular interval, OS shifts reference bits (in PTE) into counters (and clear reference bits)
 - Replacement: Pick the page with the “smallest counter”
- How many bits are enough?
 - In practice 8 bits are quite good
- Pros: Require one reference bit, small counter/page
- Cons: Require looking at many counters (or sorting)



Implementing LRU: software

- A doubly linked list of pages
- Every time page is referenced, move it to the front of the list
- Page replacement: remove the page from back of list
 - Avoid scanning of all pages
- Problem: too expensive
 - Requires 6 pointer updates for each page reference info
 - High contention on multiprocessor



Thinking

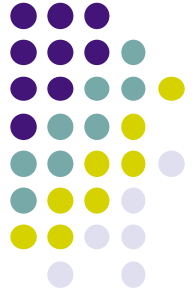
- Performance design philosophy:
- Make common case fast!
- Amadhl's Law

$$\text{Overall Speedup} = \frac{1}{(1 - F) + \frac{F}{S}}$$

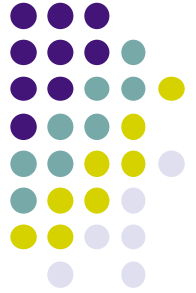
F = The fraction enhanced

S = The speedup of the enhanced fraction

Factors that affect paging performance



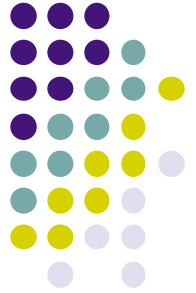
- # of memory misses
 - So far, have talked about algorithms to reduce misses (increase hit rate)
- Cost of a memory miss (replacement)



Page out on critical path?

- If no free page pool, page in has to wait till page out is finished
 - Page fault handling time = proc. overhead + 2 * I/Os
- There is a chance of paged out page being referenced soon

Page buffering techniques



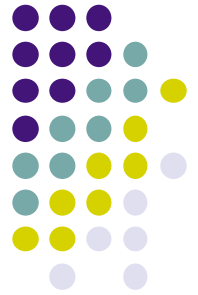
1. System maintains a pool of free pages
 - When a page fault occurs, victim page chosen as before
 - But desired page paged into a free page right away before victim page paged out
 - Dirty victim page written out when disk is idle
2. Also remember what was in free pages - maybe reused before reallocated

Possible replacement strategies



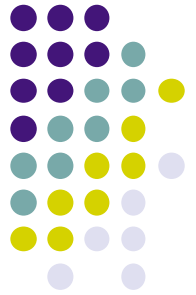
- Global replacement:
 - All pages from all processes are lumped into a single replacement pool
 - Most flexibility, least (performance) isolation
- Local replacement
 - Per-process replacement:
 - Each process has a separate pool of pages
 - Per-user replacement:
 - Lump all processes for a given user 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?

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
- Btw, what is the optimal strategy here?
 - MRU
- 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 can lead to vicious cycle

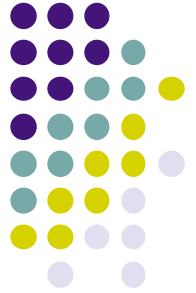


- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
 - low CPU utilization
 - OS thinks that it needs to increase the degree of multiprogramming (actual behavior of early paging systems)
 - another process added to the system
 - page fault rate goes even higher

Vicious
Cycle



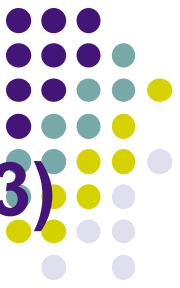
Demand paging and thrashing



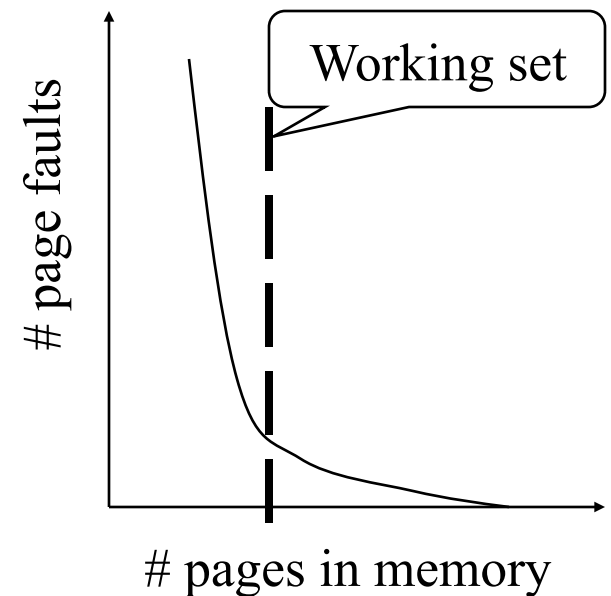
- Why does demand paging work?
 - Data reference exhibits locality
- Why does thrashing occur?
 - Σ size of locality $>$ total memory size

Working Set Model –

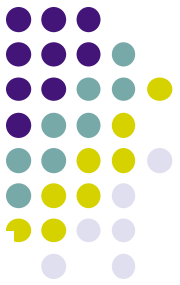
by Peter Denning (Purdue CS head, 79-83)



- An informal definition:
 - Working set: The collection of pages that a process is working within a time interval, 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 algorithm

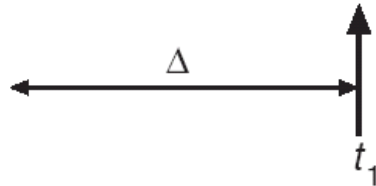


Working Sets

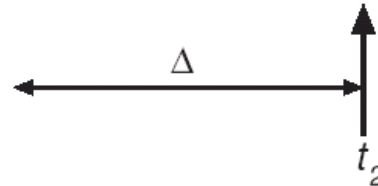


page reference table

... 2 6 1 5 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...



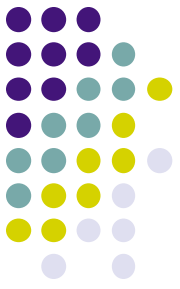
$WS(t_1) = \{1, 2, 5, 6, 7\}$



$WS(t_2) = \{3, 4\}$

- The working set size is *num of* pages in the working set
 - the number of pages touched in the interval $[t-\Delta+1..t]$.
- The working set size changes with program locality.
 - during periods of poor locality, you reference more pages.
 - Within that period of time, you will have a larger working set size.
- Goal: keep WS for each process in memory.

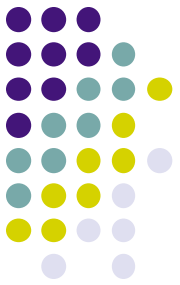
Working Set replacement algorithm



- 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**

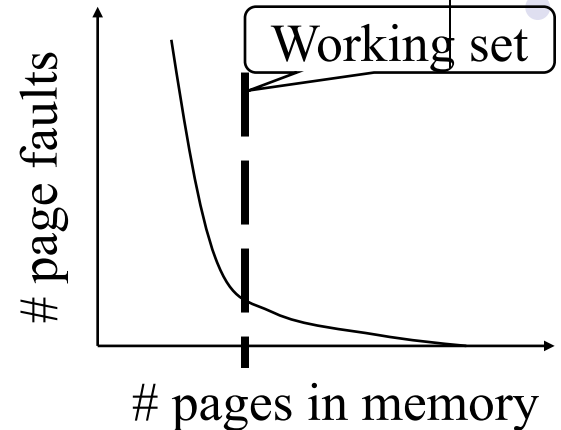
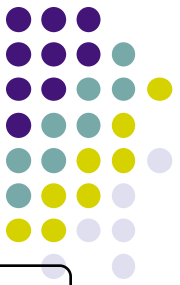
Working Set Clock Algorithm

(assume reference bit + modified bit)



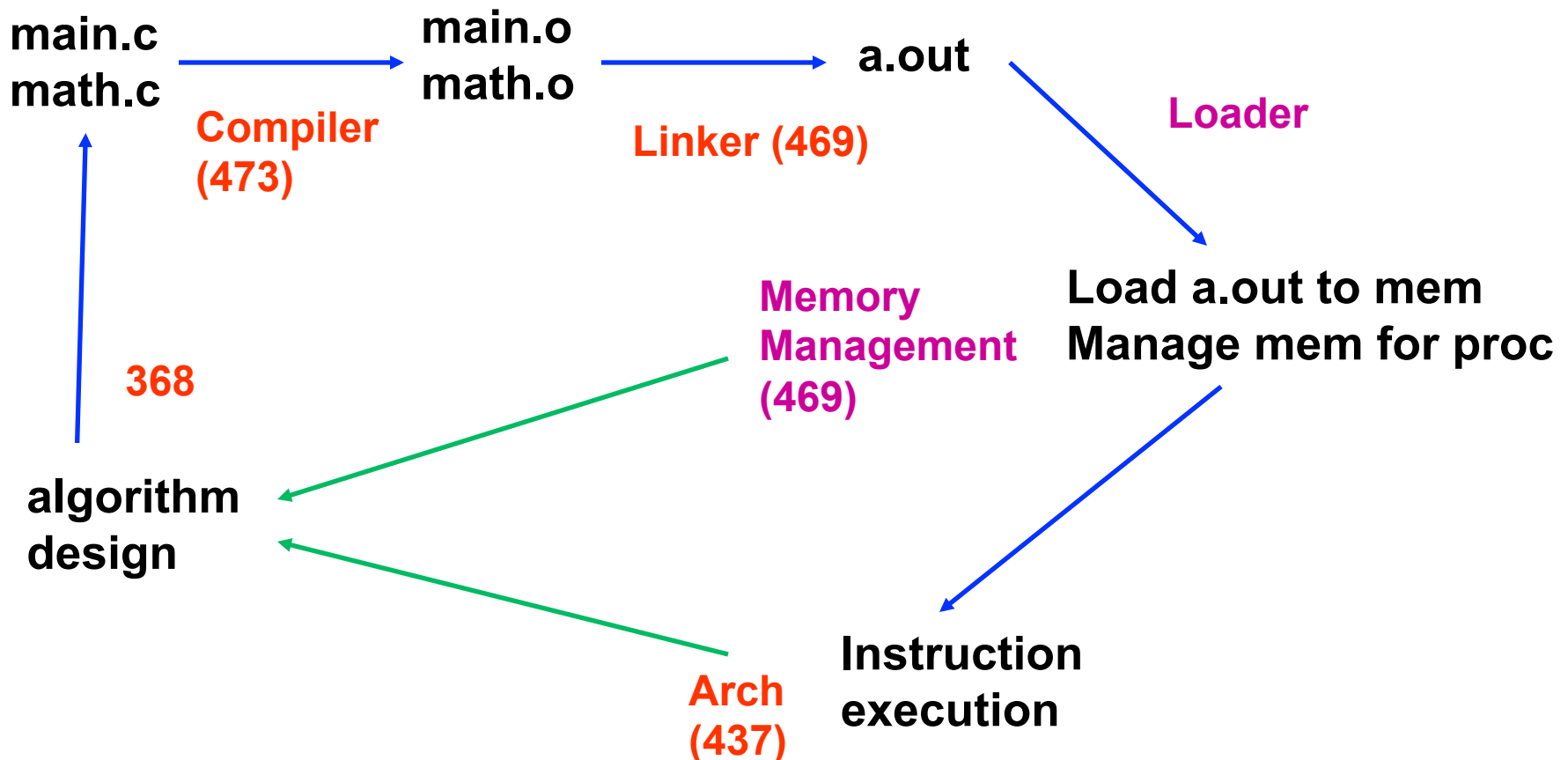
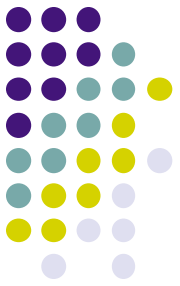
- Upon page fault, follow the clock hand
- If the reference bit is 1, set reference bit to 0, set the current time for the page and go to the next
- If the reference bit is 0, check “last use time”
 - If page used within Δ , go to the next
 - If page not used within Δ and modify bit is 1
 - Schedule 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



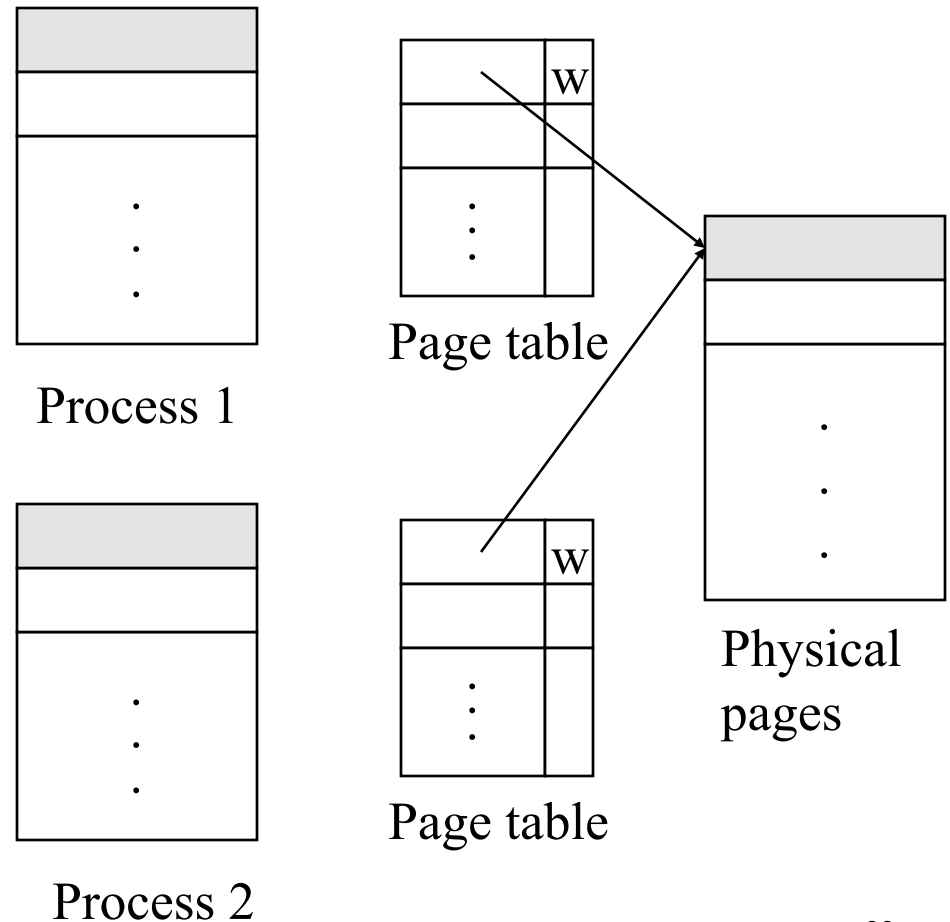
- What should Δ be?
 - What if it is too large?
 - What if it is too small?
- How many jobs need to be scheduled in order to keep CPU busy?
 - Too few \rightarrow cannot keep CPU busy if all doing I/O
 - Too many \rightarrow their WS may exceed memory

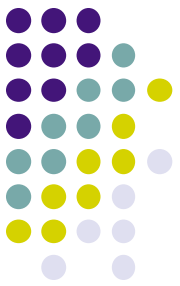
Connecting the dots ... closing the loop



Shared Memory

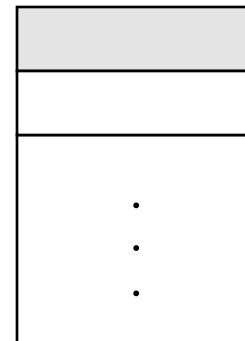
- How do two processes share memory under paging?
 - PTEs pointing to same phys addr



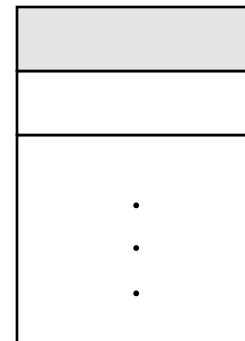


With Shared Memory

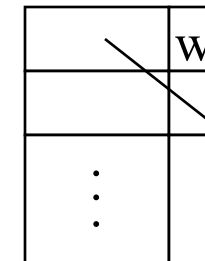
- How to destroy a virtual address space?
 - Reference count
- How to swap out/in?
 - Link all PTEs
 - Operation on all entries
- How to pin/unpin?
 - Link all PTEs
 - Reference count



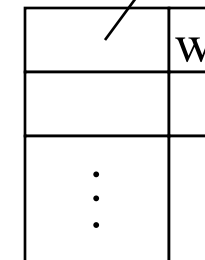
Process 1



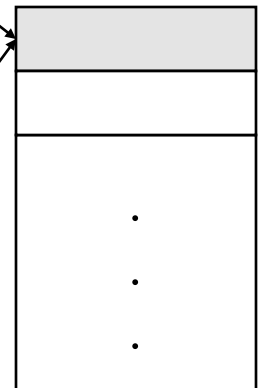
Process 2



Page table

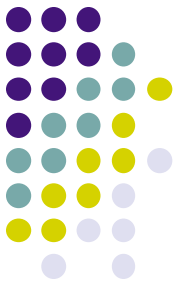


Page table



Physical
pages

[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()?

Copy-On-Write

- Child's virtual address uses the same page mapping as parent's
 - Make all pages read-only (both parent and child)
- On a read, nothing happens
- On a write (either parent or child), generates an access fault
 - map to a new page frame
 - copy the page over
 - restart the instruction
 - the other process marks the page not shared (writeable)

Used in Win2k,
Linux Solaris2
in duplicating processes

