

CS162
Operating Systems and
Systems Programming
Lecture 15

Page Allocation and
Replacement

October 24, 2005

Prof. John Kubiawicz

<http://inst.eecs.berkeley.edu/~cs162>

Review: Demand Paging Mechanisms

- PTE helps us implement demand paging
 - Valid \Rightarrow Page in memory, PTE points at physical page
 - Not Valid \Rightarrow Page not in memory; use info in PTE to find it on disk when necessary
- Suppose user references page with invalid PTE?
 - Memory Management Unit (MMU) traps to OS
 - » Resulting trap is a "Page Fault"
 - What does OS do on a Page Fault?:
 - » Choose an old page to replace
 - » If old page modified ("D=1"), write contents back to disk
 - » Change its PTE and any cached TLB to be invalid
 - » Load new page into memory from disk
 - » Update page table entry, invalidate TLB for new entry
 - » Continue thread from original faulting location
 - TLB for new page will be loaded when thread continued!
 - While pulling pages off disk for one process, OS runs another process from ready queue
 - » Suspended process sits on wait queue

Cache

10/24/05

Kubiawicz CS162 ©UCB Fall 2005

Lec 15.2

Review: Software-Loaded TLB

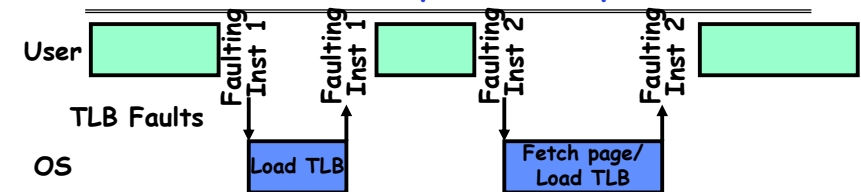
- MIPS/Snake/Nachos TLB is loaded by software
 - High TLB hit rate \Rightarrow ok to trap to software to fill the TLB, even if slower
 - Simpler hardware and added flexibility: software can maintain translation tables in whatever convenient format
- How can a process run without hardware TLB fill?
 - Fast path (TLB hit with valid=1):
 - » Translation to physical page done by hardware
 - Slow path (TLB hit with valid=0 or TLB miss)
 - » Hardware receives a "TLB Fault"
 - What does OS do on a TLB Fault?
 - » Traverse page table to find appropriate PTE
 - » If valid=1, load page table entry into TLB, continue thread
 - » If valid=0, perform "Page Fault" detailed previously
 - » Continue thread
- Everything is transparent to the user process:
 - It doesn't know about paging to/from disk
 - It doesn't even know about software TLB handling

10/24/05

Kubiawicz CS162 ©UCB Fall 2005

Lec 15.3

Review: Transparent Exceptions



- Hardware must help out by saving:
 - Faulting instruction and partial state
 - Processor State: sufficient to restart user thread
 - » Save/restore registers, stack, etc
- Precise Exception \Rightarrow state of the machine is preserved as if program executed up to the offending instruction
 - All previous instructions **completed**
 - Offending instruction and all following instructions act **as if they have not even started**
 - Difficult with pipelining, out-of-order execution, ...
 - **MIPS takes this position**
- Modern techniques for out-of-order execution and branch prediction help implement precise interrupts

10/24/05

Kubiawicz CS162 ©UCB Fall 2005

Lec 15.4

Goals for Today

- **Page Replacement Policies**
 - Clock Algorithm
 - Nth chance algorithm
 - Second-Chance-List Algorithm
- **Page Allocation Policies**
- **Working Set/Thrashing**

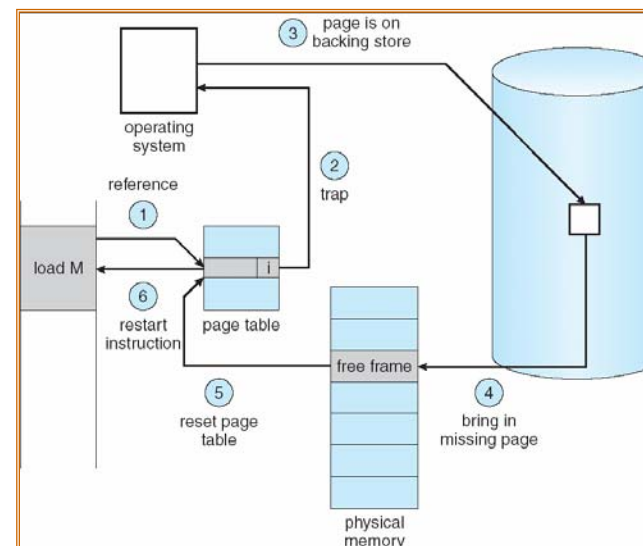
Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne

10/24/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 15.5

Steps in Handling a Page Fault



10/24/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 15.6

Demand Paging Example

- Since Demand Paging like caching, can compute average access time! ("Effective Access Time")
 - $EAT = \text{Hit Rate} \times \text{Hit Time} + \text{Miss Rate} \times \text{Miss Time}$
- Example:
 - Memory access time = 200 nanoseconds
 - Average page-fault service time = 8 milliseconds
 - Suppose p = Probability of miss, $1-p$ = Probability of hit
 - Then, we can compute EAT as follows:

$$EAT = (1 - p) \times 200\text{ns} + p \times 8 \text{ ms}$$

$$= (1 - p) \times 200\text{ns} + p \times 8,000,000\text{ns}$$

$$= 200\text{ns} + p \times 7,999,800\text{ns}$$
- If one access out of 1,000 causes a page fault, then $EAT = 8.2 \mu\text{s}$:
 - This is a slowdown by a factor of 40!
- What if want slowdown by less than 10%?
 - $200\text{ns} \times 1.1 < EAT \Rightarrow p < 2.5 \times 10^{-6}$
 - This is about 1 page fault in 400000!

10/24/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 15.7

What Factors Lead to Misses?

- **Compulsory Misses:**
 - Pages that have never been paged into memory before
 - How might we remove these misses?
 - » Prefetching: loading them into memory before needed
 - » Need to predict future somehow! More later.
- **Capacity Misses:**
 - Not enough memory. Must somehow increase size.
 - Can we do this?
 - » One option: Increase amount of DRAM (not quick fix!)
 - » Another option: If multiple processes in memory: adjust percentage of memory allocated to each one!
- **Conflict Misses:**
 - Technically, conflict misses don't exist in virtual memory, since it is a "fully-associative" cache
- **Policy Misses:**
 - Caused when pages were in memory, but kicked out prematurely because of the replacement policy
 - How to fix? Better replacement policy

10/24/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 15.8

Page Replacement Policies

- Why do we care about Replacement Policy?
 - Replacement is an issue with any cache
 - Particularly important with pages
 - » The cost of being wrong is high: must go to disk
 - » Must keep important pages in memory, not toss them out
- FIFO (First In, First Out)
 - Throw out oldest page. Be fair - let every page live in memory for same amount of time.
 - Bad, because throws out heavily used pages instead of infrequently used pages
- MIN (Minimum):
 - Replace page that won't be used for the longest time
 - Great, but can't really know future...
 - Makes good comparison case, however
- RANDOM:
 - Pick random page for every replacement
 - Typical solution for TLB's. Simple hardware
 - Pretty unpredictable - makes it hard to make real-time guarantees

10/24/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 15.9

Replacement Policies (Con't)

- LRU (Least Recently Used):
 - Replace page that hasn't been used for the longest time
 - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
 - Seems like LRU should be a good approximation to MIN.
- How to implement LRU? Use a list!

 - On each use, remove page from list and place at head
 - LRU page is at tail
- Problems with this scheme for paging?
 - Need to know immediately when each page used so that can change position in list...
 - Many instructions for each hardware access
- In practice, people approximate LRU (more later)

10/24/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 15.10

Example: FIFO

- Suppose we have 3 page frames, 4 virtual pages, and following reference stream:
 - A B C A B D A D B C B
- Consider FIFO Page replacement:

Ref:	A	B	C	A	B	D	A	D	B	C	B
Page:											
1	A					D				C	
2		B					A				
3			C						B		

- FIFO: 7 faults.
- When referencing D, replacing A is bad choice, since need A again right away

10/24/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 15.11

Example: MIN

- Suppose we have the same reference stream:
 - A B C A B D A D B C B
- Consider MIN Page replacement:

Ref:	A	B	C	A	B	D	A	D	B	C	B
Page:											
1	A									C	
2		B									
3			C			D					

- MIN: 5 faults
- Where will D be brought in? Look for page not referenced farthest in future.
- What will LRU do?
 - Same decisions as MIN here, but won't always be true!

10/24/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 15.12

Administrivia

- Exam is graded: grades should be in glookup
 - Average: 71.2
 - Standard Dev: 12.3
- If you are 2 or more standard-deviations below the mean, you need to do better:
 - You are in danger of getting a D or F
 - Feel free to come to talk with me
- Solutions to the Midterm are up on the Handouts page
 - They were up there Friday, but don't know if people noticed
- Project 2 autograder:
 - Will be run a couple of times today and tomorrow
 - More times on Wednesday
 - Yet more times on Thursday
- Web mirror:
 - Problem with links after last class: people couldn't get notes
 - » Sorry about that! I am the right person to complain to...
 - There is a mirror of the course web site at: <http://www.cs.berkeley.edu/~kubitron/cs162>

10/24/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 15.13

When will LRU perform badly?

- Consider the following: A B C D A B C D A B C D
- LRU Performs as follows (same as FIFO here):

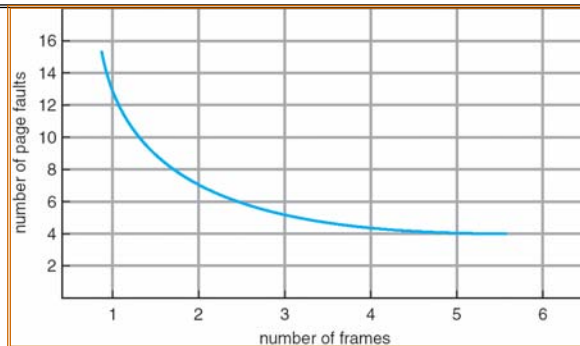
Ref:	A	B	C	D	A	B	C	D	A	B	C	D
Page:												
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

- Every reference is a page fault!
- MIN Does much better:

Ref:	A	B	C	D	A	B	C	D	A	B	C	D
Page:												
1	A									B		
2		B					C					
3			C	D								

10

Graph of Page Faults Versus The Number of Frames



- One desirable property: When you add memory the miss rate goes down
 - Does this always happen?
 - Seems like it should, right?
- No: BeLady's anomaly
 - Certain replacement algorithms (FIFO) don't have this obvious property!

10/24/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 15.15

Adding Memory Doesn't Always Help Fault Rate

- Does adding memory reduce number of page faults?
 - Yes for LRU and MIN
 - Not necessarily for FIFO! (Called Belady's anomaly)

Ref:	A	B	C	D	A	B	E	A	B	C	D	E
Page:												
1	A			D			E					
2		B			A					C		
3			C			B					D	
Page:												
1	A						E				D	
2		B						A				E
3			C						B			
4				D						C		

- After adding memory:
 - With FIFO, contents can be completely different
 - In contrast, with LRU or MIN, contents of memory with X pages are a subset of contents with X+1 Page

10/24/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 15.16

Implementing LRU

- **Perfect:**
 - Timestamp page on each reference
 - Keep list of pages ordered by time of reference
 - Too expensive to implement in reality for many reasons
- **Clock Algorithm:** Arrange physical pages in circle with single clock hand
 - Approximate LRU (approx to approx to MIN)
 - Replace **an** old page, not **the oldest** page
- **Details:**
 - Hardware "use" bit per physical page:
 - » Hardware sets use bit on each reference
 - » If use bit isn't set, means not referenced in a long time
 - » Nachos hardware sets use bit in the TLB; you have to copy this back to page table when TLB entry gets replaced
 - On page fault:
 - » Advance clock hand (not real time)
 - » Check use bit: 1→used recently; clear and leave alone
 - 0→selected candidate for replacement
 - Will always find a page or loop forever?
 - » Even if all use bits set, will eventually loop around⇒FIFO

10/24/05

Kubiatawicz CS162 ©UCB Fall 2005

Lec 15.17

Clock Algorithm: Not Recently Used



Single Clock Hand:

Advances only on page fault!
Check for pages not used recently
Mark pages as not used recently

- What if hand moving slowly?
 - Good sign or bad sign?
 - » Not many page faults and/or find page quickly
- What if hand is moving quickly?
 - Lots of page faults and/or lots of reference bits set
- One way to view clock algorithm:
 - Crude partitioning of pages into two groups: young and old
 - Why not partition into more than 2 groups?

10/24/05

Kubiatawicz CS162 ©UCB Fall 2005

Lec 15.18

Nth Chance version of Clock Algorithm

- **Nth chance algorithm:** Give page N chances
 - OS keeps counter per page: # sweeps
 - On page fault, OS checks use bit:
 - » 1→clear use and also clear counter (used in last sweep)
 - » 0→increment counter; if count=N, replace page
 - Means that clock hand has to sweep by N times without page being used before page is replaced
- How do we pick N?
 - Why pick large N? Better approx to LRU
 - » If N ~ 1K, really good approximation
 - Why pick small N? More efficient
 - » Otherwise might have to look a long way to find free page
- What about dirty pages?
 - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
 - Common approach:
 - » Clean pages, use N=1
 - » Dirty pages, use N=2 (and write back to disk when N=1)

10/24/05

Kubiatawicz CS162 ©UCB Fall 2005

Lec 15.19

Clock Algorithms: Details

- Which bits of a PTE entry are useful to us?
 - **Use:** Set when page is referenced; cleared by clock algorithm
 - **Modified:** set when page is modified, cleared when page written to disk
 - **Valid:** ok for program to reference this page
 - **Read-only:** ok for program to read page, but not modify
 - » For example for catching modifications to code pages!
- Do we really need hardware-supported "modified" bit?
 - No. Can emulate it (BSD Unix) using read-only bit
 - » Initially, mark all pages as read-only, even data pages
 - » On write, trap to OS. OS sets modified bit, marks page as read-write.
 - » Whenever page comes back in from disk, mark read-only

10/24/05

Kubiatawicz CS162 ©UCB Fall 2005

Lec 15.20

Clock Algorithms Details (continued)

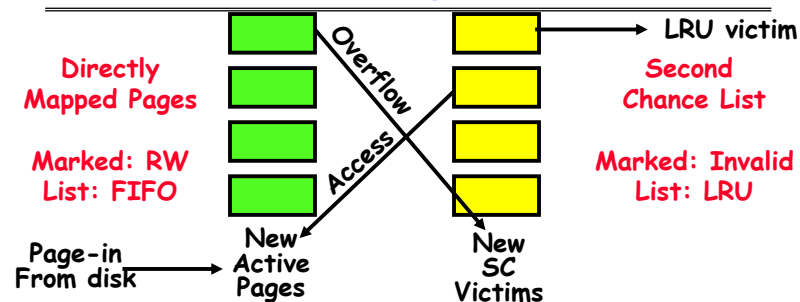
- Do we really need a hardware-supported "use" bit?
 - No. Can emulate it similar to above:
 - » Mark all pages as invalid, even if in memory
 - » On read to invalid page, trap to OS
 - » OS sets use bit, and marks page read-only
 - Get modified bit in same way as previous:
 - » On write, trap to OS (either invalid or read-only)
 - » Set use and modified bits, mark page read-write
 - When clock hand passes by, reset use and modified bits and mark page as invalid again
- Remember, however, that clock is just an approximation of LRU
 - Can we do a better approximation, given that we have to take page faults on some reads and writes to collect use information?
 - Need to identify an old page, not oldest page!
 - Answer: second chance list

10/24/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 15.21

Second-Chance List Algorithm (VAX/VMS)



- Split memory in two: Active list (RW), SC list (Invalid)
- Access pages in Active list at full speed
- Otherwise, Page Fault
 - Always move overflow page from end of Active list to front of Second-chance list (SC) and mark invalid
 - Desired Page On SC List: move to front of Active list, mark RW
 - Not on SC list: page in to front of Active list, mark RW; page out LRU victim at end of SC list

10/24/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 15.22

Second-Chance List Algorithm (con't)

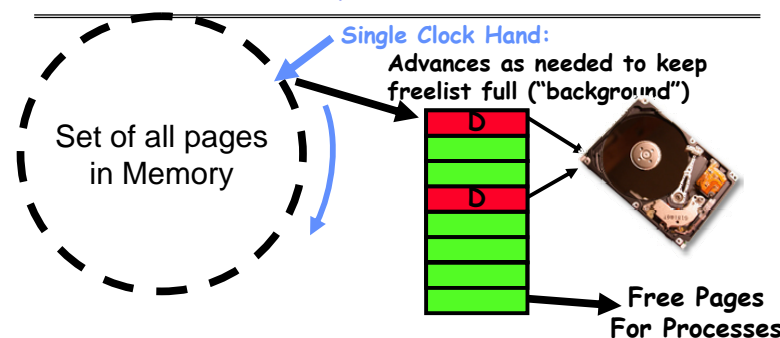
- How many pages for second chance list?
 - If 0 \Rightarrow FIFO
 - If all \Rightarrow LRU, but page fault on every page reference
- Pick intermediate value. Result is:
 - Pro: Few disk accesses (page only goes to disk if unused for a long time)
 - Con: Increased overhead trapping to OS (software / hardware tradeoff)
- With page translation, we can adapt to any kind of access the program makes
 - Later, we will show how to use page translation / protection to share memory between threads on widely separated machines
- Question: why didn't VAX include "use" bit?
 - Strecker (architect) asked OS people, they said they didn't need it, so didn't implement it
 - He later got blamed, but VAX did OK anyway

10/24/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 15.23

Free List



- Keep set of free pages ready for use in demand paging
 - Freelist filled in background by Clock algorithm or other technique ("Pageout demon")
 - Dirty pages start copying back to disk when enter list
- Like VAX second-chance list
 - If page needed before reused, just return to active set
- Advantage: Faster for page fault
 - Can always use page (or pages) immediately on fault

10/24/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 15.24

Demand Paging (more details)

- Does software-loaded TLB need use bit?
Two Options:
 - Hardware sets use bit in TLB; when TLB entry is replaced, software copies use bit back to page table
 - Software manages TLB entries as FIFO list; everything not in TLB is Second-Chance list, managed as strict LRU
- Core Map
 - Page tables map virtual page → physical page
 - Do we need a reverse mapping (i.e. physical page → virtual page)?
 - » Yes. Clock algorithm runs through page frames. If sharing, then multiple virtual-pages per physical page
 - » Can't push page out to disk without invalidating all PTEs

10/24/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 15.25

Allocation of Page Frames (Memory Pages)

- How do we allocate memory among different processes?
 - Does every process get the same fraction of memory?
Different fractions?
 - Should we completely swap some processes out of memory?
- Each process needs *minimum* number of pages
 - Want to make sure that all processes **that are loaded into memory** can make forward progress
 - Example: IBM 370 - 6 pages to handle SS MOVE instruction:
 - » instruction is 6 bytes, might span 2 pages
 - » 2 pages to handle *from*
 - » 2 pages to handle *to*
- Possible Replacement Scopes:
 - **Global replacement** - process selects replacement frame from set of all frames; one process can take a frame from another
 - **Local replacement** - each process selects from only its own set of allocated frames

10/24/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 15.26

Fixed/Priority Allocation

- **Equal allocation (Fixed Scheme):**
 - Every process gets same amount of memory
 - Example: 100 frames, 5 processes ⇒ process gets 20 frames
- **Proportional allocation (Fixed Scheme)**
 - Allocate according to the size of process
 - Computation proceeds as follows:
 - s_i = size of process p_i and $S = \sum s_i$
 - m = total number of frames
 - a_i = allocation for $p_i = \frac{s_i}{S} \times m$
- **Priority Allocation:**
 - Proportional scheme using priorities rather than size
 - » Same type of computation as previous scheme
 - Possible behavior: If process p_i generates a page fault, select for replacement a frame from a process with lower priority number
- Perhaps we should use an adaptive scheme instead???
 - What if some application just needs more memory?

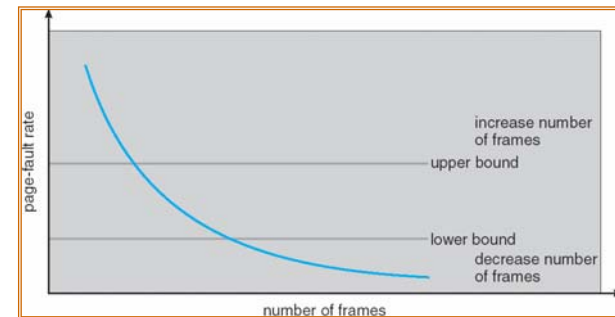
10/24/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 15.27

Page-Fault Frequency Allocation

- Can we reduce Capacity misses by dynamically changing the number of pages/application?



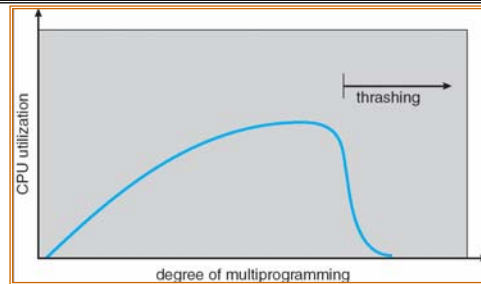
- Establish "acceptable" page-fault rate
 - If actual rate too low, process loses frame
 - If actual rate too high, process gains frame
- Question: What if we just don't have enough memory?

10/24/05

Kubiatowicz CS162 ©UCB Fall 2005

Lec 15.28

Thrashing



- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
 - low CPU utilization
 - operating system spends most of its time swapping to disk
- **Thrashing** \equiv a process is busy swapping pages in and out
- Questions:
 - How do we detect Thrashing?
 - What is best response to Thrashing?

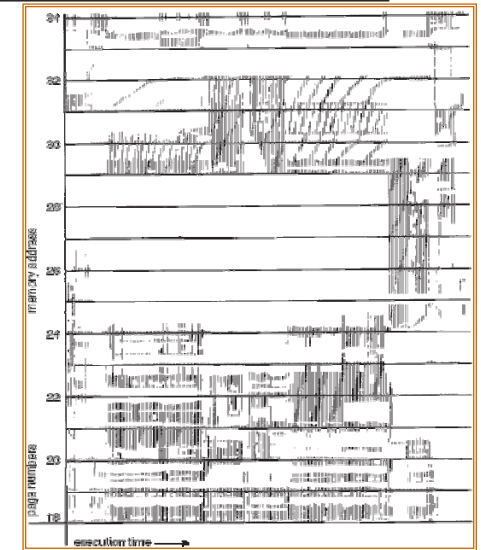
10/24/05

Kubiatowicz CS162 @UCB Fall 2005

Lec 15.29

Locality In A Memory-Reference Pattern

- Program Memory Access Patterns have temporal and spatial locality
 - Group of Pages accessed along a given time slice called the "Working Set"
 - Working Set defines minimum number of pages needed for process to behave well
- Not enough memory for Working Set \Rightarrow Thrashing
 - Better to swap out process?

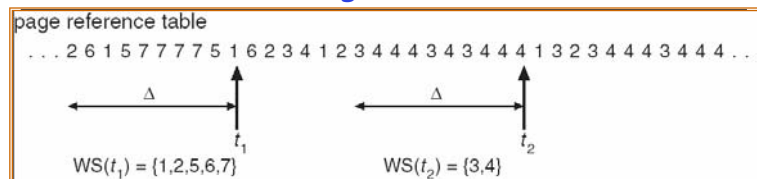


10/24/05

Kubiatowicz CS162 @UCB Fall 2005

Lec 15.30

Working-Set Model



- $\Delta \equiv$ working-set window \equiv fixed number of page references
 - Example: 10,000 instructions
- WS_i (working set of Process P_i) = total set of pages referenced in the most recent Δ (varies in time)
 - if Δ too small will not encompass entire locality
 - if Δ too large will encompass several localities
 - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \sum |WS_i| \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing
 - Policy: if $D > m$, then suspend one of the processes
 - This can improve overall system behavior by a lot!

10/24/05

Kubiatowicz CS162 @UCB Fall 2005

Lec 15.31

What about Compulsory Misses?

- Recall that compulsory misses are misses that occur the first time that a page is seen
 - Pages that are touched for the first time
 - Pages that are touched after process is swapped out/swapped back in
- **Clustering:**
 - On a page-fault, bring in multiple pages "around" the faulting page
 - Since efficiency of disk reads increases with sequential reads, makes sense to read several sequential pages
- **Working Set Tracking:**
 - Use algorithm to try to track working set of application
 - When swapping process back in, swap in working set

10/24/05

Kubiatowicz CS162 @UCB Fall 2005

Lec 15.32

Summary

- **Replacement policies**
 - FIFO: Place pages on queue, replace page at end
 - MIN: Replace page that will be used farthest in future
 - LRU: Replace page used farthest in past
- **Clock Algorithm: Approximation to LRU**
 - Arrange all pages in circular list
 - Sweep through them, marking as not "in use"
 - If page not "in use" for one pass, then can replace
- **Nth-chance clock algorithm: Another approx LRU**
 - Give pages multiple passes of clock hand before replacing
- **Second-Chance List algorithm: Yet another approx LRU**
 - Divide pages into two groups, one of which is truly LRU and managed on page faults.
- **Working Set:**
 - Set of pages touched by a process recently
- **Thrashing: a process is busy swapping pages in and out**
 - Process will thrash if working set doesn't fit in memory
 - Need to swap out a process