

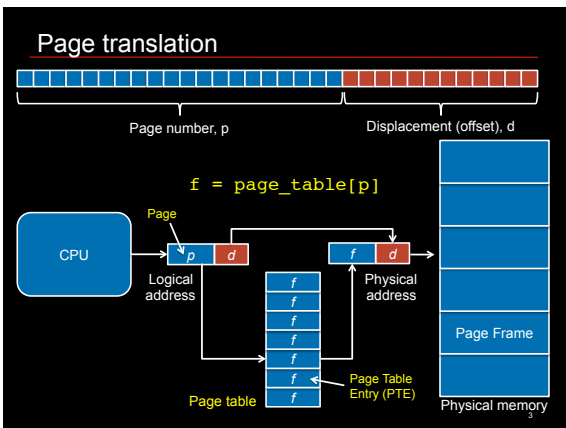
Operating Systems Design
 9. Memory Management: Part 2

Paul Krzyzanowski
 pxk@cs.rutgers.edu

1

Recap

2



Page table

- One page table per process
- Stores page permissions:
 - Read-only
 - No-execute
 - Dirty
 - Modified
 - Others (e.g., secure or privileged mode access)

4

Hardware Implementation: TLB

- Cache frequently-accessed pages
 - Translation lookaside buffer (TLB)
 - Associative memory: key (page #) and value (frame #)
- TLB is on-chip & fast ... but small (64 – 1,024 entries)
- TLB miss: result not in the TLB
 - Need to do page table lookup in memory
- Hit ratio = % of lookups that come from the TLB

5

Real-Time Considerations

- Avoid page table lookup
 - Or run CPU without virtual addressing
- Pin high-priority real-time process memory into TLB (if possible)

6

Page-Based Virtual Memory Benefits

- Simplify memory management for multiprogramming
 - Allow **discontiguous allocation**
 - MMU give the illusion of contiguous allocation
- Allow a process to feel that it **has more memory** than it really has
 - Also: process can have greater address space than system memory
- **Memory Protection**
 - Each process' address space is separate from others
 - MMU allows pages to be protected:
 - Writing, execution, kernel vs. user access

Accessing memory

- Process makes *virtual* address references for all memory access
- MMU converts to physical address via a per-process table
 - Page number → Page frame number
 - Basic info stored in a PTE (page table entry):
 - Valid flag?
 - Page frame number?
 - Permissions
 - **Page fault** if not a valid reference
- Most CPUs support:
 - **Virtual addressing mode** and **Physical addressing mode**
 - CPU starts in physical mode ... someone has to set up page tables
 - Divide address space into user & kernel spaces

Kernel's View

- A process sees a flat linear address space
 - Accessing regions of memory mapped to the kernel causes a page fault
- Kernel's view:
 - Address split into two parts
 - User part: changes with context switches
 - Kernel part: remains constant
 - Split is configurable:
 - 32-bit x86: PAGE_OFFSET: 3GB for process + 1 GB kernel

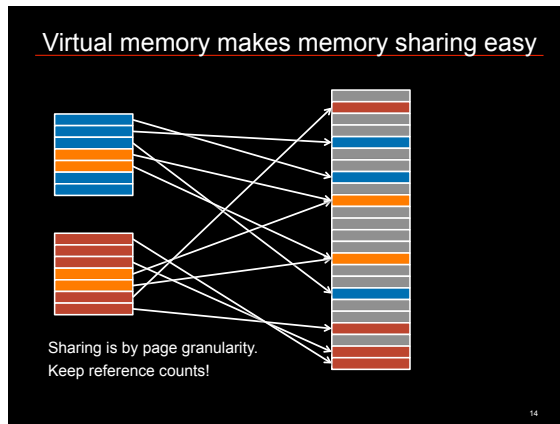
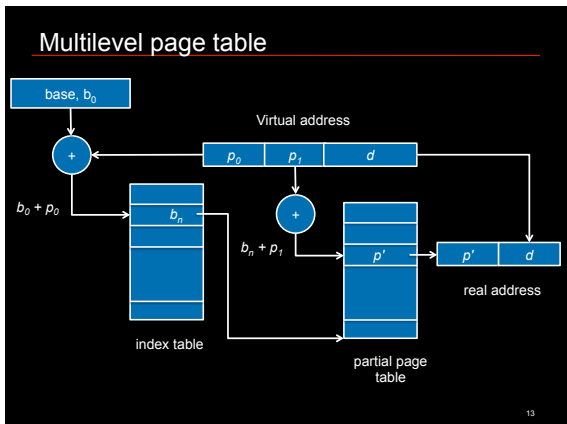
Page allocator

- With VM, processes can use discontiguous pages
- Kernel may need contiguous buffers
 - free_area: keep track of lists of free pages
 - 1st element: free single pages
 - 2nd element: free blocks of 2 pages
 - 3rd element: free blocks of 4 pages
 - Etc.
- Buddy algorithm
 - Try to get the best usable allocation unit
 - If not available, get the next biggest one & split
 - Coalesce upon free

Sample memory map per process

Multilevel (Hierarchical) page tables

- Most processes use only a small part of their address space
- Keeping an entire page table is wasteful
- E.g., 32-bit system with 4KB pages: 20-bit page table
 - 1,048,576 entries in a page table

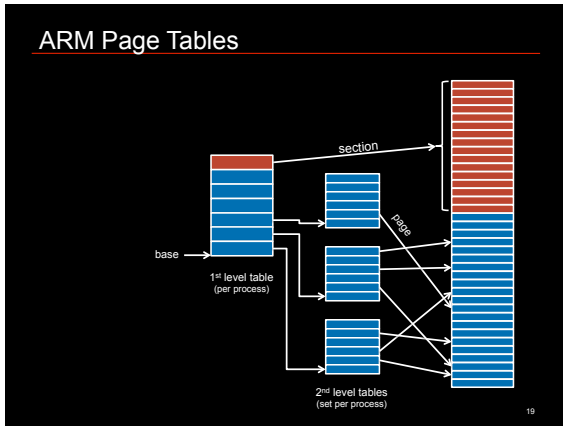


MMU Example: ARM

- ### ARMv7-A architecture
- Cortex-A8
 - iPhone 3GS, iPod Touch 3G, Apple A4 processor in iPad, Droid X, Droid 2, etc.)
 - Cortex-A9
 - TI OMAP 44xx series, Apple A5 processor in iPad 2

- ### Pages
- Four page (block) sizes:
- Supersections: 16MB memory blocks
 - Sections: 1MB memory blocks
 - Large pages 64KB memory blocks
 - Small pages 4KB memory blocks

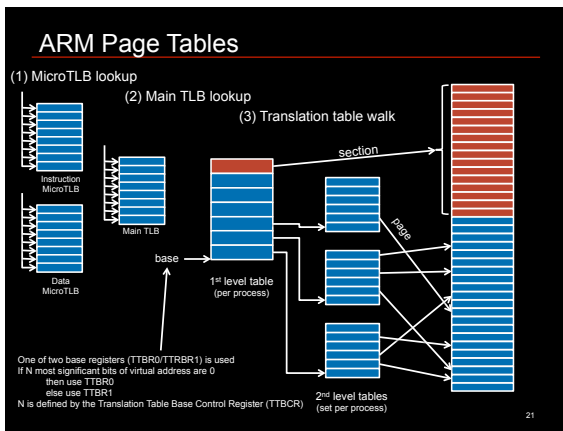
- ### Two levels of tables
- **First level table**
 - Base address, descriptors, and translation properties for sections and supersections (1 MB & 16 MB blocks)
 - Translation properties and pointers to a **second level table** for large and small pages (4 KB and 64 KB pages)
 - **Second level tables (aka page tables)**
 - Each contains base address and translation properties for small and large pages
- Benefit: a large region of memory can be mapped using a single entry in the TLB (e.g., OS)



TLB

- 1st level: **MicroTLB** – one each for instruction & data sides
 - 32 entries (10 entries in older v6 architectures)
 - Address Space Identifier (**ASID**) [8 bits] and Non-Secure Table Identifier (NSTID) [1 bit]; entries can be global
 - Fully associative; one-cycle lookup
 - Lookup checks protection attributes: may signal Data Abort
 - Replacement either Round-Robin (default) or Random
- 2nd level: **Main TLB** – catches cache misses from microTLBs
 - 8 fully associative entries (may be locked) + 64 low associative entries
 - variable number of cycles for lookup
 - lockdown region of 8 entries (important for real-time)
 - Entries are globally mapped or associated ASID and NSTID

20



Translation flow for a section (1 MB)

Virtual address: 31 20 19 0
 Table index (12 bits) | Section offset (20 bits)

Physical section = Read [Translation table base + table index]
 Physical address = physical section : section offset

Real address: 31 20 19 0
 Physical section (12 bits) | Section offset (20 bits)

22

Translation flow for a supersection (16 MB)

Virtual address: 31 24 23 12 11 0
 Table index (8 bits) | Supersection offset (24 bits)

Supersection base address, Extended base address =
 Read [Translation table base + table index]
 Real address = Extended base address : physical section : section offset

Real address: 39 32 31 24 23 0
 Extended base address (8 bits) | Supersection base address (8 bits) | Supersection offset (24 bits)
 40 bit address

23

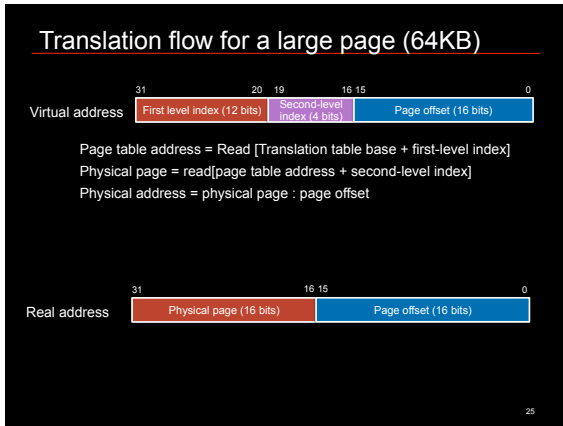
Translation flow for a small page (4KB)

Virtual address: 31 20 19 12 11 0
 First level index (12 bits) | Second-level index (8 bits) | Page offset (12 bits)

Page table address = Read [Translation table base + first-level index]
 Physical page = read[page table address + second-level index]
 Real address = physical page : page offset

Real address: 31 12 11 0
 Physical page (20 bits) | Page offset (12 bits)

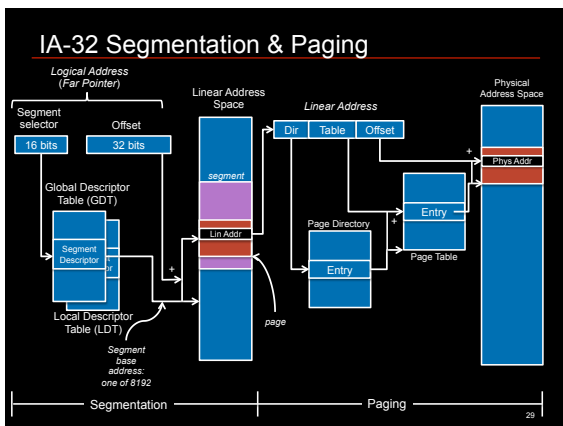
24



- ### Memory Protection & Control
- Domains
 - Clients execute & access data within a domain. Each access is checked against access permissions for each memory block
 - Memory region attributes
 - Execute never
 - Read-only, read/write, no access
 - Privileged read-only, privileged & user read-only
 - Non-secure (is this secure memory or not?)
 - Sharable (is this memory shared with other processors)
 - Strongly ordered (memory accesses must occur in program order)
 - Device/shared, device/non-shared
 - Normal/shared, normal/non-shared
 - Signal Memory Abort if permission is not valid for access

MMU Example: x86-64

- ### IA-32 Memory Models
- Flat memory model
 - Linear address space
 - Single, contiguous address space
 - Segmented memory model
 - Memory appears as a group of independent address spaces: segments (code, data, stack, etc.)
 - Logical address = {segment selector, offset}
 - 16,383 segments; each segment can be up to 2^{32} bytes
 - Real mode
 - 8086 model
 - Segments up to 64KB in size
 - maximum address space: 2^{20} bytes



- ### Segment protection
- S flag in segment descriptor identifies *code* or *data* segment
 - Accessed
 - has the segment been accessed since the last time the OS cleared the bit?
 - Dirty
 - Has the page been modified?
 - Data
 - Write-enabled
 - Read-only or read/write?
 - Expansion direction
 - Expand down (e.g., for stack); dynamically changing the segment limit causes space to be added to the bottom of the stack
 - Code
 - Execute only, execute/read (e.g., constants in code segment)
 - Conforming:
 - Execution can continue even if privilege level is elevated

IA-32 Paging

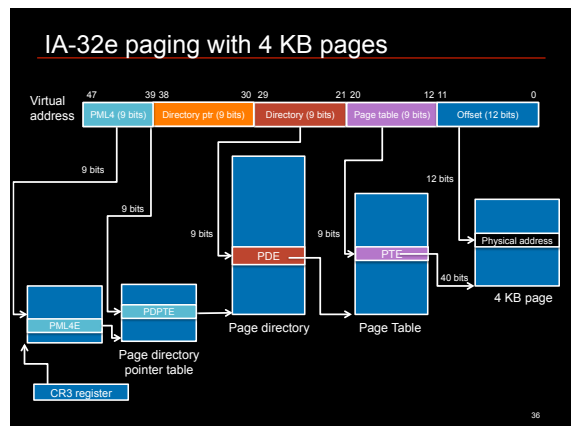
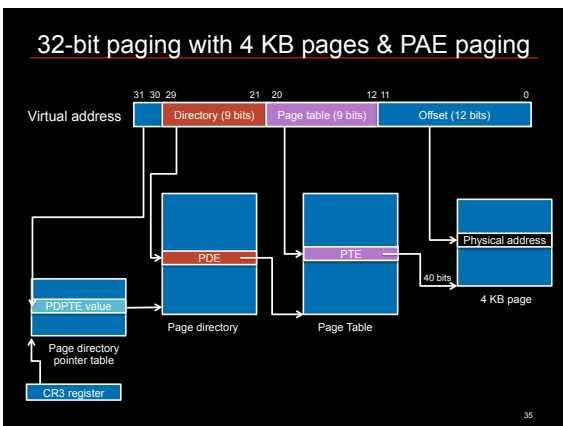
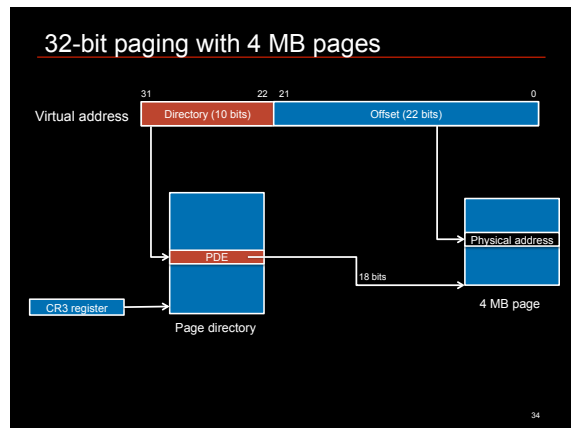
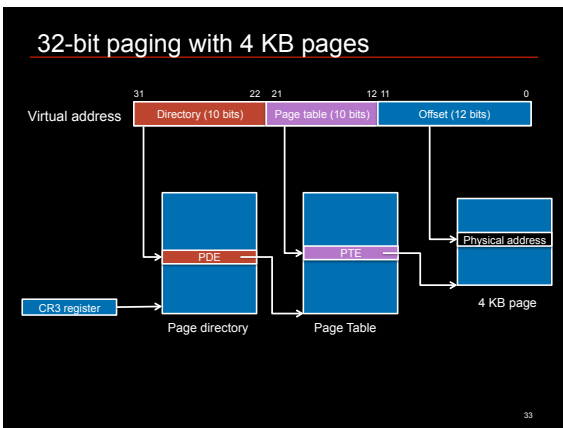
- 32-bit registers, 36-bit address space (64 GB)
 - Physical Address Extension (PAE)
 - Bit 5 of control register CR4
 - 52 bit physical address support (4 PB)
 - Only a 4 GB address space may be accessed at one time
 - Page Size Extensions (PSE-36)
 - 36-bit page size extension
 - Support 4 MB pages

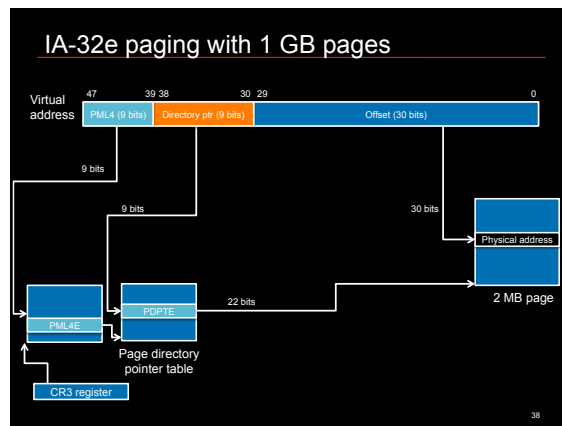
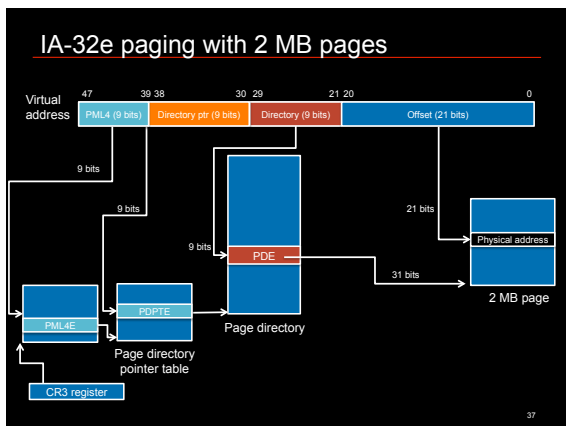
31

Intel 64-bit mode

- Segments supported in IA-32 emulation mode
 - Mostly disabled for 64-bit mode
 - 64-bit base addresses where used
- Three paging modes
 - 32-bit paging
 - 32-bit virtual address; 32-40 bit physical
 - 4 KB or 4 MB pages
 - PAE
 - 32-bit virtual addresses; up to 52-bit physical address
 - 4 KB or 2 MB pages
 - IA-32e paging
 - 48-bit virtual addresses; up to 52-bit physical address
 - 4 KB, 2 MB, or 1 GB pages

32





- ### TLBs on the Core i7
- 4 KB pages
 - Instruction TLB: 128 entries per core
 - Data TLB: 64 entries
 - Core 2 Duo: 16 entries TLB0; 256 entries TLB1
 - Atom: 64-entry TLB, 16-entry PDE
 - Second-level unified TLB
 - 512 entries
- 39

- ### Managing Page Tables
- Linux: architecture independent (mostly)
 - Abstract structures to model 4-level page tables
 - Actual page tables are stored in a machine-specific manner
- 40

- ### Recap
- Fragmentation is a non-issue
 - Page table
 - Page table entry (PTE)
 - Multi-level page tables
 - Inverted page table
 - Segmentation
 - Segmentation + Paging
 - Memory protection
 - Isolation of address spaces
 - Access control defined in PTE
- 41

Demand Paging

42

Executing a program

- Allocate memory + stack and load the entire program from memory (including linked libraries)
- Then execute it

43

Executing a program

- Allocate memory + stack and load the entire program from memory (including linked libraries)
- Then execute it

We don't need to do this!

44

Demand Paging

- Load pages into memory only as needed
 - On first access
 - Pages that are never used never get loaded
- Use valid/invalid bit in page table entry
 - Valid: the page is in memory ("valid" mapping)
 - Invalid: out of bounds access or page is not in memory
- Invalid memory access generates a *page fault*

45

Demand Paging: At Process Start

- Open executable file
- Set up memory map (stack & text/data/bss)
 - But don't load anything!
- Load first page & allocate initial stack page
- Run it!

46

Memory Mapping

- Executable files & libraries must be brought into a process' virtual address space
 - File is *mapped* into the process' memory
 - As pages are referenced, page frames are allocated & pages are loaded
- `vm_area_struct`
 - Defines regions of virtual memory
 - Used in setting page table entries
 - Start of VM region, end of region, access rights
- Several of these are created for each mapped image
 - Executable code, initialized data, uninitialized data

47

Demand Paging: Page Fault Handling

- Soon the process will access an address without a valid page
 - OS gets a page fault
- What happens?
 - Kernel searches a tree structure of memory allocations for the process to see if the faulting address is valid
 - If not valid, send a SEGV signal to the process
 - Is the type of access valid for the page?
 - Send a signal
 - We have a valid page but it's not in memory

48

Demand Paging: Getting a Page

- The page we need is either in the executable file or in a swap file
 - If PTE is not valid but page # is present
 - The page we want has been saved to a swap file
 - Page # in the PTE tells us the location in the file
 - If the PTE is not valid and no page #
 - Load the page from the program file from the disk
- Read page into physical memory
 - Find a free page frame (evict one if necessary)
 - Read the page: This takes time: context switch & block
 - Update page table for the process
 - Restart the process at the instruction that faulted

49

Page Replacement

- If the address space used by all processes + OS \leq physical memory, we're ok
- Otherwise:
 - Make room: discard or store a page onto the disk
 - If the page came from a file & was not modified
 - Discard ... we can always get it
 - If the page is dirty, it must be saved in a **swap file**.

50

Cost

- Handle page fault exception: ~ 400 usec
- Disk seek & read: ~ 10 msec
- Memory access: ~ 100 ns
- Page fault degrades performance by around 100,000!!
- Avoid page faults!
 - If we want $< 10\%$ degradation of performance, we can have just one page fault per 1,000,000 memory accesses

51

Page replacement

- We need a good replacement policy for good performance

52

FIFO Replacement

- First In, First Out
- Good
 - May get rid of initialization code or other code that's no longer used
- Bad
 - May get rid of a page holding frequently used global variables

53

Least Recently Used (LRU)

- Timestamp a page when it is accessed
- When we need to remove a page, search for the one with the oldest timestamp
- Nice algorithm but...
 - Timestamping is a pain – we can't do it with the MMU!

54

Not Frequently Used Replacement

- Each PTE has a reference bit
- Keep a counter for each page frame
- At each clock interrupt:
 - Add the reference bit of each frame to its counter
 - Clear reference bit
- To evict a page, choose the frame with the lowest counter
- Problem
 - No sense of time: a page that was used a lot a long time ago may still have a high count
 - Updating counters is expensive

55

Clock (Second Chance)

- Arrange physical pages in a logical circle (circular queue)
 - Clock hand points to first frame
- Paging hardware keeps 1 *use* bit per frame
 - Set *use* bit on memory reference
 - If it's not set then the frame hasn't been used for a while
- On page fault:
 - Advance clock hand
 - Check *use* bit
 - If 1, it's been used recently – clear & advance
 - If 0, evict this page

56

Nth Chance Replacement

- Similar to Second Chance
- Maintain a counter along with a use bit
- On page fault:
 - Advance clock hand
 - Check use bit
 - If 1, clear and set counter to 0
 - If 0, increment counter. If counter < N, go on. Else evict
- Better approximation of LRU

57

Kernel Swap Daemon

- *kswapd* on Linux
- Anticipate problems
- Decides whether to shrink caches if page count is low
 - Page cache, buffer cache
 - Evict pages from page frames

58

Demand paging summary

- Allocate page table
 - Map kernel memory
 - Initialize stack
 - Memory-map text & data from executable program (& libraries)
 - But don't load!
- Load pages on demand (first access)
 - When we get a page fault

59

Summary: If we run out of free page frames

- Free some page frames
 - Discard pages that are mapped to a file or
 - Move some pages to a **swap** file
- Clock algorithm
- Anticipate need for free page frames
 - *kswapd* – kernel swap daemon

60

Multitasking Considerations

61

Supporting multitasking

- Multiple address spaces can be loaded in memory
- A CPU register points to the current page table
- OS changes the register set when context switching
- Performance increased with Address Space ID in TLB

62

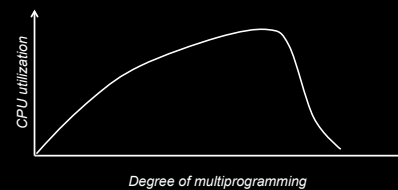
Working Set

- Keep active pages in memory
- A process needs its working set in memory to perform well
 - Working set = set of pages that have been referenced in the last window of time
 - **Spatial locality**
 - Size of working set varies during execution
- More processes in a system:
 - Good: increase throughput; chance that some process is available to run
 - Bad: **thrashing**: processes do not have enough page frames available to run without paging

63

Thrashing

- Locality
 - Process migrates from one locality (working set) to another
- Thrashing
 - Occurs when sum of all working sets > total memory



64

Resident Set Management

- How many pages of a process do we bring in?
- Resident set can be fixed or variable
- Replacement scope: global or local
 - Global: process can pick a replacement from *all* frames
- Variable allocation with global scope:
 - Simple
 - Replacement policy may not take working sets into consideration
- Variable allocation with local scope
 - More complex
 - Modify resident size to approximate working set size

65

Working Set Model

- Δ : *working set window*:
 - Amount of elapsed time while the process was actually executing (e.g., count of memory references)
- Approximates locality of a program
- WSS_i : working set size of process P_i
 - WSS_i = set of pages in most recent Δ page references
- System-wide demand for frames

$$D = \sum WSS_i$$
- If $D >$ total memory size, then we get thrashing

66

Page fault frequency

- Too small a working set causes a process to thrash
- Monitor page fault frequency per process
 - If too high, the process needs more frames
 - If too low, the process may have too many frames

67

Dealing with thrashing

- If all else fails ...
- Suspend a process(es)
 - Lowest priority, Last activated, smallest resident set, ...?
- Swapping:
 - Move an entire process onto the disk: no pages in memory
 - Process must be re-loaded to run

68

The End

69