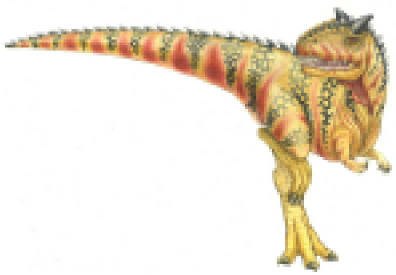
Chapter 10: Virtual Memory



Outline

- Background

Objectives

- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

- Define virtual memory and describe its benefits.
- Illustrate how pages are loaded into memory using demandpaging.
- Apply the FIFO, optimal, and LRU page-replacement algorithms.
- Describe the working set of a process and explain howit isrelatedto program locality.
- Describe how Linux, Windows 10, and Solaris managevirtual memory.
- Design a virtual memory manager simulation in the Cprogramminglanguage.

Background

- Code needs to be in memory to execute, but entire programrarelyused

• Error code, unusual routines, large data structures ▪ Entire program code not needed at same time ▪ Consider ability to execute partially-loaded program• Program no longer constrained by limits of physical memory

- Each program takes less memory while running ->moreprograms run at the same time
  - 4 Increased CPU utilization and throughput with noincreaseinresponse time or turnaround time
- Less I/O needed to load or swap programs into memory->eachuser program runs faster

Virtual memory

- **Virtual memory** – separation of user logical memory fromphysical memory

- Only part of the program needs to be in memory for execution
- Logical address space can therefore be much larger thanphysical address space
  - Allows address spaces to be shared by several processes
- Allows for more efficient process creation
- More programs running concurrently
- Less I/O needed to load or swap processes

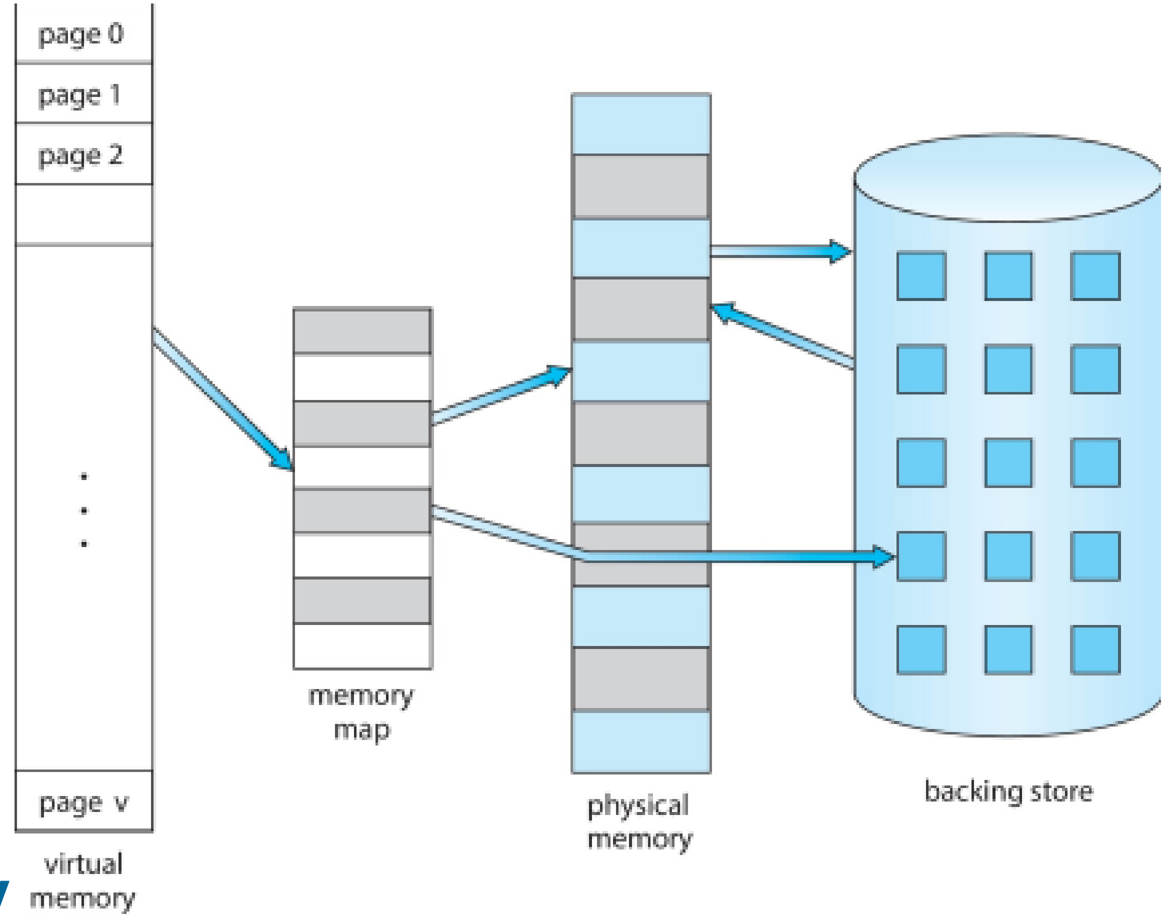
Virtual memory(Cont.)

- **Virtual address space** – logical view of how process is stored in memory

- Usually start at address 0, contiguous addresses until end of space
- Meanwhile, physical memory organized in page frames
- MMU must map logical to physical

- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

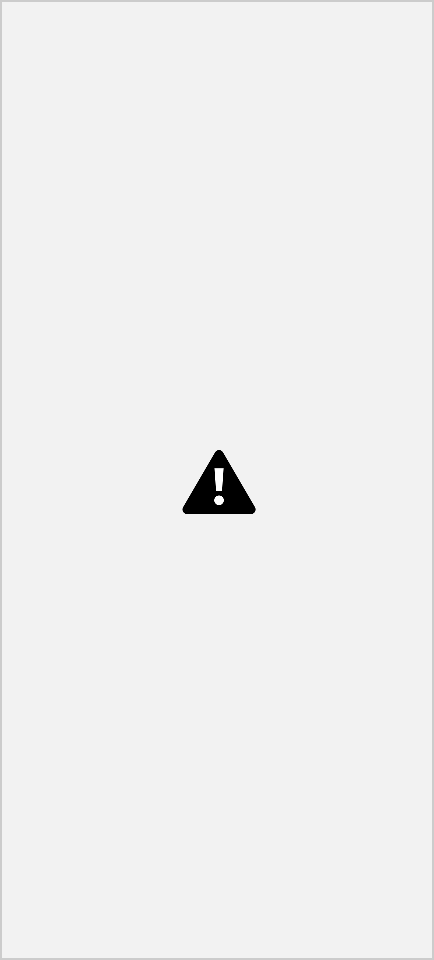
Virtual Memory That is Larger



ThanPhysicalMemory

Virtual-addressSpace

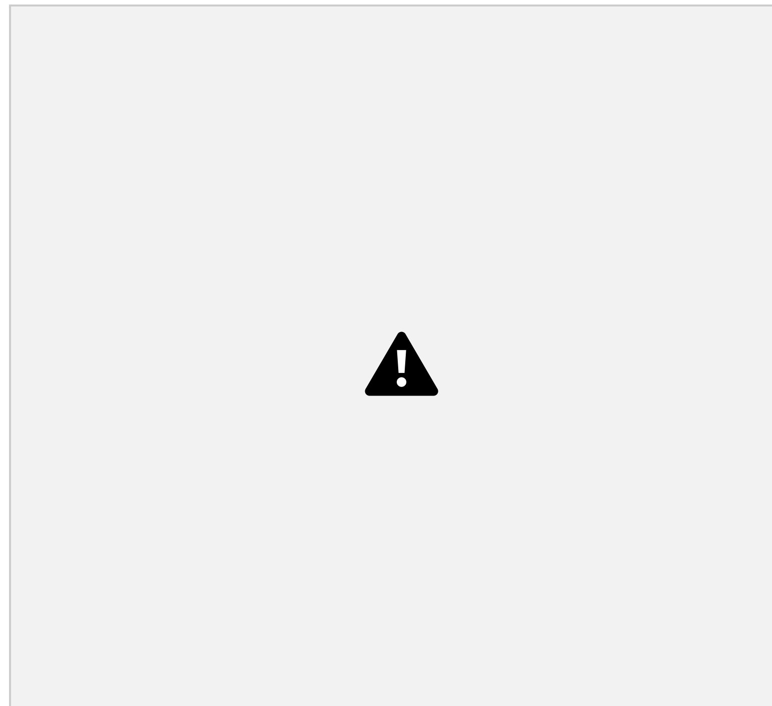
- Usually design logical address space for the stack to start at Max logical address and grow "down" while heap grows "up"
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc.
- System libraries shared via mapping into virtual address space
  - Shared memory by mapping pages read- write into virtual address space
  - Pages can be shared during `fork()`, speeding process creation



Shared Library Using Virtual Memory

DemandPaging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)



- invalid reference ⇒ abort
  - Not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

Basic Concepts

- With swapping, the pager guesses which pages will beusedbefore swapping them out again
- How to determine that set of pages?
- Need new MMU functionality to implement demand paging ▪ If pages needed are already **memory resident**
  - No difference from non demand-paging ▪ If page needed and not memory resident
  - Need to detect and load the page into memory fromstorage 4 Without changing program behavior

    4 Without programmer needing to change code ▪ Use page table with valid-invalid bit (see chapter 9)

Page table withValid-InvalidBit

- With each page table entry a valid–invalid bit is associated(**v** ⇒ in-memory, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

- During MMU address translation, if valid–invalid bit in thepage table entry is **i**
  - ⇒ page fault

Page Table When Some PagesAreNotin Main Memory

Steps in HandlingPageFault

1. If there is a reference to a page, first reference to that page will trap to operating system • Page fault
2. Operating system looks at another table to decide: • Invalid reference ⇒ abort • Just not in memory (go to step 3)
3. Find free frame (what if there is none?)
4. Swap page into frame via scheduled disk operation 5. Reset tables to indicate page now in memory Set validation bit = **v**
6. Restart the instruction that caused the page fault

Steps in Handling a PageFault(Cont.)

Aspects of DemandPaging

- **Pure demand paging:** start process with *no* pages in memory
- OS sets instruction pointer to first

instruction of process, non-memory-resident -> page fault

• And for every other process pages on first access ▪ Actually, a given instruction could access multiple pages->multiple page faults

• Consider fetch and decode of instruction which adds2numbers from memory and stores result back to memory▪ Hardware support needed for demand paging • Page table with valid / invalid bit

• Secondary memory (swap device with **swap space**) • Instruction restart

Free-FrameList

▪ When a page fault occurs, the operating systemmust bringthedesired page from secondary storage into main memory. ▪ Most operating systems maintain a **free-frame list** -- apool of free frames for satisfying such requests.

- Operating system typically allocate free frames using atechniqueknown as **zero-fill-on-demand** -- the content of the frameszeroed-out before being allocated.
- When a system starts up, all available memory is placedonthefree-frame list.

Stages in Demand Paging–WorseCase

1. Trap to the operating system 2. Save the user registers and process state 3. Determine that the interrupt was a page fault 4. Check that the page reference was legal and determinethelocationofthe page on the disk

5. Issue a read from the disk to a free frame:

a) Wait in a queue for this device until the read request isservicedb) Wait for the device seek and/or latency time c) Begin the transfer of the page to a free frame

Stages in DemandPaging(Cont.)

6. While waiting, allocate the CPU to some other user 7. Receive an interrupt from the disk I/O subsystem(I/Ocompleted)

8. Save the registers and process state for the other user 9. Determine that the interrupt was from the disk 10. Correct the page table and other tables to

showpageisnow in memory

11. Wait for the CPU to be allocated to this process again12. Restore the user registers, process state, and newpagetable, and then resume the interrupted instruction

Performance of DemandPaging

▪ Three major activities

- Service the interrupt – careful coding means just several hundred instructions needed
- Input the page from disk – lots of time
- Restart the process – again just a small amount of time

- Page Fault Rate $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

EAT = (1 – p) x memory access

+ p (page fault overhead

+ swap page out

+ swap page in)

Demand PagingExample

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)

= (1 – p) x 200 + p x 8,000,000

= 200 + p x 7,999,800

- If one access out of 1,000 causes a page fault, then EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!

- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025

    4 one page fault in every 400,000 memory accesses

Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes toinitially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the pagecopied
- COW allows more efficient process creation as only modified pagesarecopied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand**pages

- Pool should always have free frames for fast demand page execution
  - 4 Don’t want to have to free a frame as well as other processingon page fault
- Why zero-out a page before allocating it?

- `vfork()` variation on `fork()` system call has parent suspend andchildusing copy-on-write address space of parent
  - Designed to have child call `exec()`
  - Very efficient

Before Process 1 ModifiesPageC

After Process 1 ModifiesPageC

What Happens if ThereisnoFreeFrame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not reallyinuse,page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result inminimumnumber of page faults
- Same page may be brought into memory several times

Page Replacement

- Prevent **over-allocation** of memory by modifying page-fault serviceroutine to include page replacement
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers–onlymodified pages are written to disk
- Page replacement completes separation between logical memoryandphysical memory – large virtual memory can be providedonasmallerphysical memory

Need For PageReplacement

Basic Page Replacement

1. Find the location of the desired page on disk 2. Find a free frame:

   - If there is a free frame, use it

   - If there is no free frame, use a page replacement algorithmto select a **victim frame**

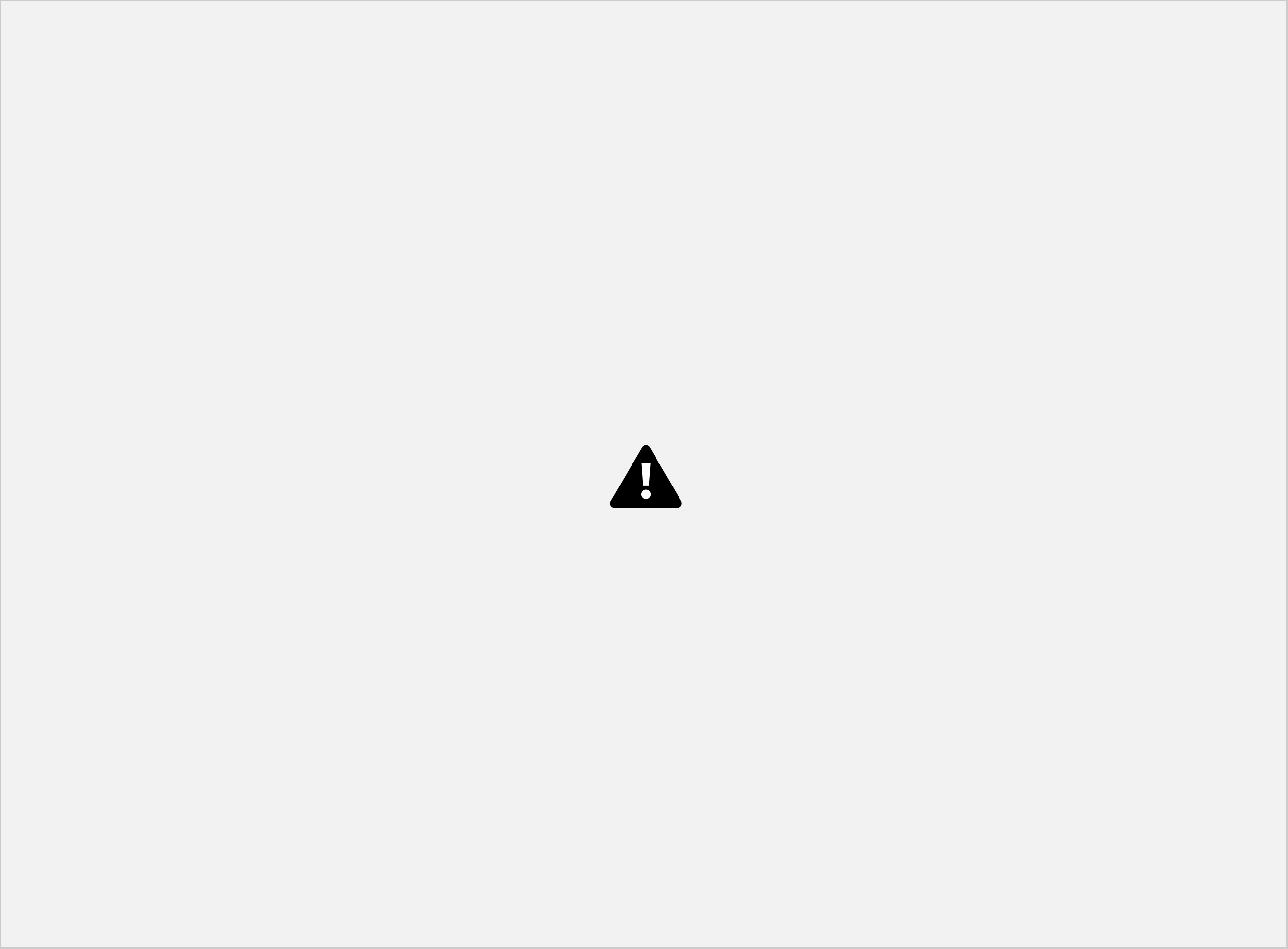
   - Write victim frame to disk if dirty

3. Bring the desired page into the (newly) free frame; updatethepageand frame tables

4. Continue the process by restarting the instruction that causedthetrap

Note now potentially 2 page transfers for page fault – increasingEAT

Page Replacement



Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines • How many frames to give each process • Which frames to replace
- **Page-replacement algorithm** • Want lowest page-fault rate on both first access andre-access ▪ Evaluate algorithm by running it on a particular string of memoryreferences (reference string) and computing the number of pagefaults on that string
  - String is just page numbers, not full addresses • Repeated access to the same page does not causeapagefault • Results depend on number of frames available
- In all our examples, the **reference string** of referencedpagenumbers is

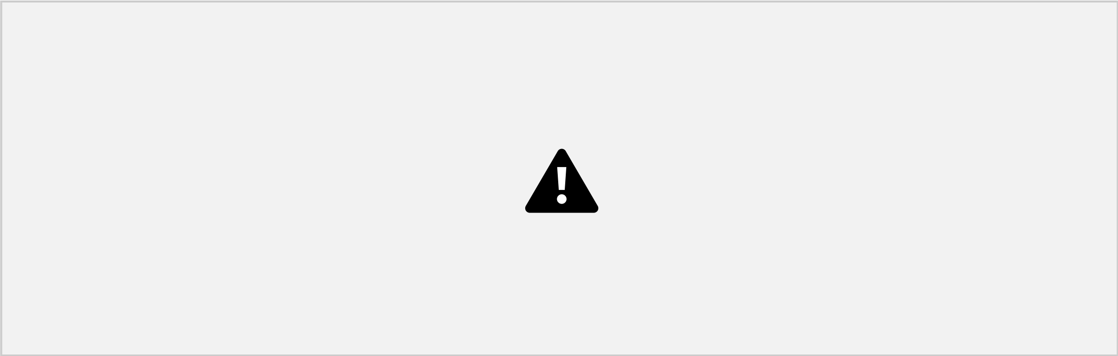
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0
,1,7,0,1

Graph of Page Faults Versus theNumberofFrames

First-In-First-Out (FIFO) Algorithm

Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**

- 3 frames (3 pages can be in memory at a time per process)



15 page faults

- How to track ages of pages?
  - Just use a FIFO queue

Belady's Anomaly

- Consider the string 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
- Graph illustrating Belady's Anomaly

Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example

- How do you know this?
  - Can’t read the future
    - Used for measuring how well your algorithmperforms

- Optimal is an example of **stack algorithms** that don’t suffer from Belady’s Anomaly

Least Recently Used(LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page

- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- LRU is another example of stack algorithms; thus it doesnot sufferfrom Belady's Anomaly

LRU Algorithm Implementation

Time-counter implementation

- Every page entry has a time-counter variable; every timeapage is referenced through this entry, copy the valueof theclock into the time-counter
- When a page needs to be changed, look at the time-countersto find smallest value
  - 4 Search through a table is needed

- Stack implementation
  - Keep a stack of page numbers in a double link form: • Page referenced:
    - 4 Move it to the top
    - 4 Requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement

Stack Implementation

- Use of a stack to record most recent page references

LRU ApproximationAlgorithms

Needs special hardware

- **Reference bit**
  - With each page associate a bit, initially = 0 • When page is referenced bit set to 1
- Replace any with reference bit = 0 (if one exists) • We do not know the order, however

LRU ApproximationAlgorithms(cont.)

Second-chance algorithm

- Generally FIFO, plus hardware-provided referencebit

Clock replacement

- If page to be replaced has
  - Reference bit = 0 -> replace it
  - Reference bit = 1 then:
    - Set reference bit 0, leave page in memory
    - Replace next page, subject to same rules

Second-chanceAlgorithm

Enhanced Second-ChanceAlgorithm

- Improve algorithm by using reference bit and modify bit (if available)in concert
- Take ordered pair (reference, modify):
  - (0, 0) neither recently used not modified – best pagetoreplace• (0, 1) not recently used but modified – not quite as good, mustwrite out before replacement
  - (1, 0) recently used but clean – probably will be usedagainsoon• (1, 1) recently used and modified – probably will beusedagainsoon and need to write out before replacement
- When page replacement called for, use the clock schemebut usethefour classes replace page in lowest non-empty class • Might need to search circular queue several times

Counting Algorithms

- Keep a counter of the number of references that have beenmadeto each page
  - Not common
- **Lease Frequently Used** (**LFU**) **Algorithm**:
  - Replaces page with smallest count
- **Most Frequently Used** (**MFU**) **Algorithm**:
  - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Page-BufferingAlgorithms

Keep a pool of free frames, always

- Then frame available when needed, not found at fault time
- Read page into free frame and select victimto evict andaddtofree pool
- When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages thereandset tonon-dirty
- Possibly, keep free frame contents intact and note what isinthem
  - If referenced again before reused, no need to loadcontentsagainfrom disk
    - Generally useful to reduce penalty if wrong victimframeselected

Page-BufferingAlgorithms

- Keep a pool of free frames which is never empty
  - Thus a frame is available when needed, not foundat fault time
  - Read page into free frame and select victimto evict andaddtofree pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages thereandset tonon-dirty
- Possibly, keep free frame contents intact and note what isinthem
  - If referenced again before reused, no need to loadcontentsagainfrom disk
  - Generally useful to reduce penalty if wrong victimframeselected

Applications andPageReplacement

All of these algorithms have OS guessing about futurepageaccess ▪ Some applications have better knowledge – i.e., databases

▪ Memory intensive applications can cause double buffering • OS keeps copy of page in memory as I/Obuffer

• Application keeps page in memory for its own work ▪ Operating system can provide direct access to the disk, gettingoutof the way of the applications

• **Raw disk** mode

▪ Bypasses buffering, locking, etc.

Allocationof Frames

- Each process needs ***minimum*** number of frames
- Example: IBM 370 – 6 pages to handle SS MOVEinstruction:
  - Instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- ***Maximum*** of course is total frames in the system
- Two major allocation schemes
  - Fixed allocation
  - Priority allocation
- Many variations

Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give eachprocess20 frames
  - Keep some as free frame buffer pool
- Proportional allocation – Allocate according to the sizeof process
  - Dynamic as degree of multiprogramming, process sizeschange

$m = 64$

$s\ p =$

size of process

$i\ i$

$s_1 = 10$

$S\ s = \sum$

i

$s_2 = 127\ a_1 = 10$

$m =$

total number of frames $137 \times 62 \approx 4$

m
S^s

i

a ~~p~~ = ~~=~~ × allocation ~~for~~

$i\ i$

$a_2 = 127$

$137 \times 62 \approx 57$

Global vs. Local Allocation

- **Global replacement** – process selects a replacement framefromthe set of all frames; one process can take a frame fromanother
  - Process execution time can vary greatly
  - Greater throughput so more commonly used
- **Local replacement** – each

process selects fromonly itsownset of allocated frames

- More consistent per-process performance
- But possibly underutilized memory
  - What if a process does not have enough frames?

ReclaimingPages

- A strategy to implement global page-replacement policy
- All memory requests are satisfied from the free-framelist, ratherthan waiting for the list to drop to zero before we beginselectingpages for replacement,
- Page replacement is triggered when the list falls

belowacertainthreshold.

- This strategy attempts to ensure there is always sufficient freememory to satisfy new requests.

Reclaiming PagesExample

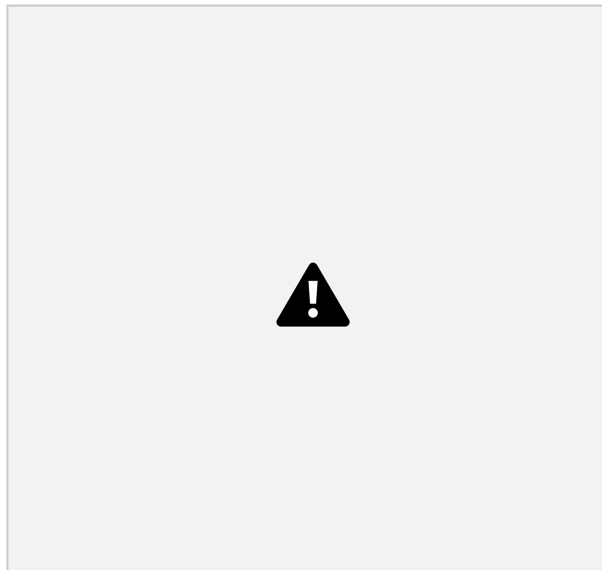
Non-UniformMemoryAccess

- So far, we

assumed that all memory accessed equally ▪ Many systems are **NUMA** – speed of access to memory varies

- Consider system boards containing CPUs and memory, interconnected over a system bus

▪ NUMA multiprocessing architecture



NUMAAccess(Cont.)

▪ Optimal performance comes

from allocating memory

"closeto"the

CPU on which the thread is scheduled

- And modifying the scheduler to schedule the threadonthesamesystem board when possible

- Solved by Solaris by creating **lgroups**

4 Structure to track CPU / Memory low latency groups4 Used my

schedule and pager

4 When possible schedule all threads of a process andallocateall memory for that process within the lgroup

Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need the replaced frame back
- This leads to:
  - Low CPU utilization
  - Operating system thinking that it needs to increasethedegree of multiprogramming
  - Another process added to the system

Thrashing(Cont.)

- **Thrashing.** A process is busy swapping pages in andout

Demand PagingandThrashing

- Why does demand paging work?

Locality model

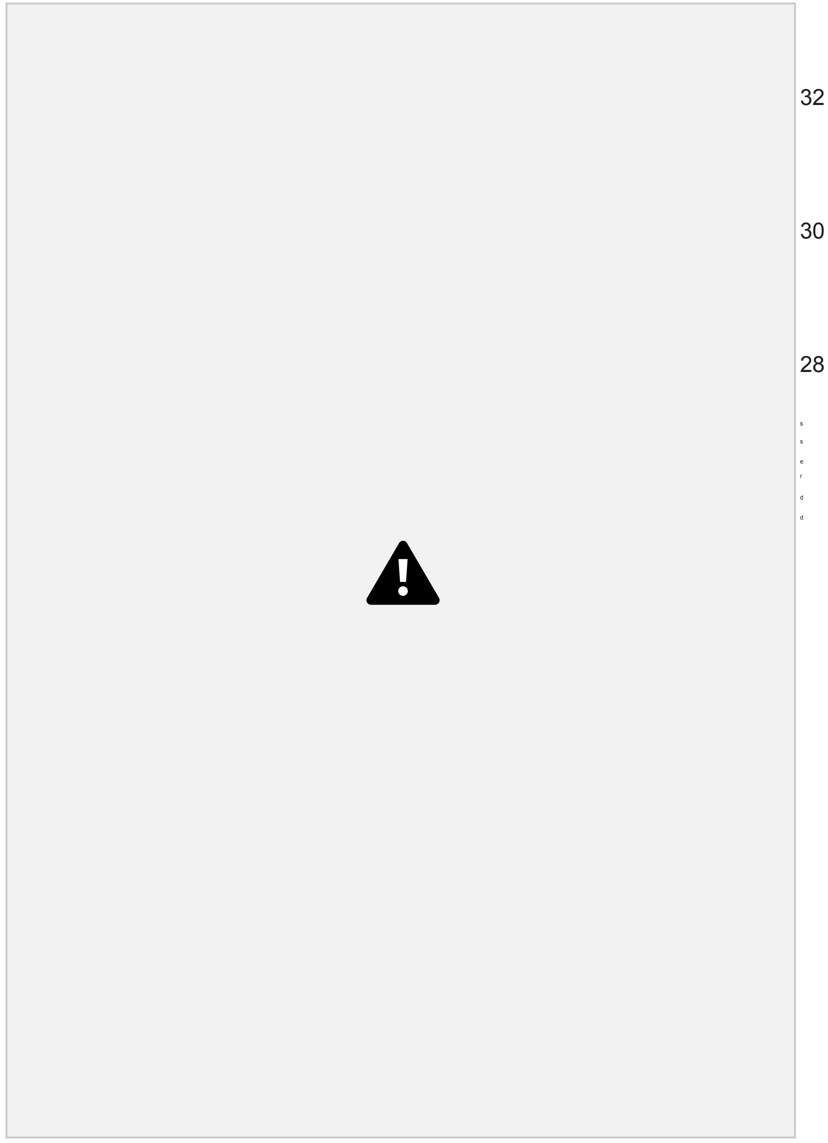
  - Process migrates from one locality to another
  - Localities may overlap

- Why does thrashing occur?

  Σ size of locality > total memory size

- To avoid trashing:
  - Calculate the Σ size of locality
  - Policy:
    4 if Σ size of locality > total memory size□suspendor swap out one of the processes
- Issue: how to calculate "Σ size of locality"

Locality In AMemory-ReferencePattern



26

24

22

20

18

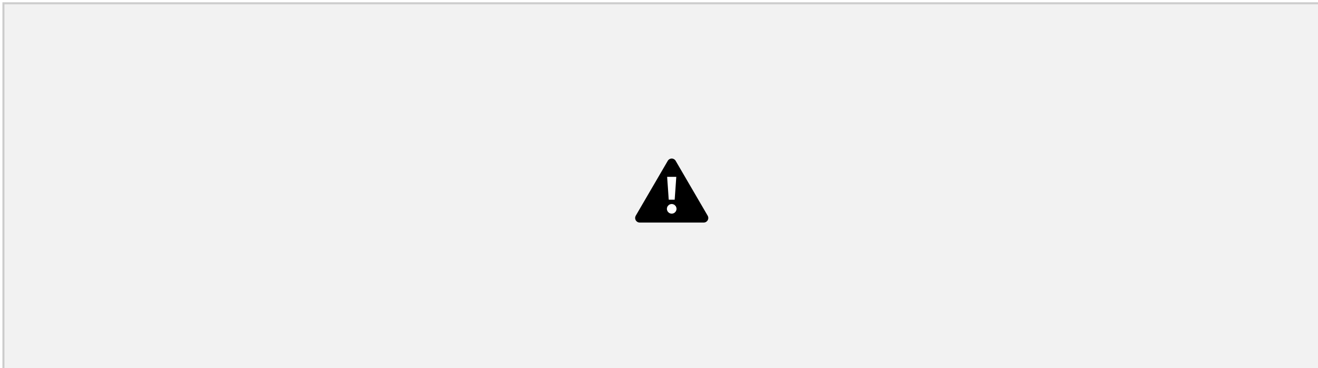
execution ~~time~~

Working-Set Model

- Δ ≡ working-set window ≡ a fixed number of page referencesExample: 10,000 instructions
- *WSS*$_i$ (working set of Process *P*$_i$) = total number of pagesreferenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass the entire locality • if Δ too large will

encompass several localities • if Δ = ∞ ⇒ will encompass entire program

- Example



Working-Set Model

- Δ ≡ working-set window ≡ a fixed number of page referencesExample: 10,000 instructions
- WSS_i (working set of Process P_i) = total number of pagesreferenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass the entire locality • if Δ too large will

encompass several localities • if $\Delta = \infty \Rightarrow$ will encompass entire program

- $D = \Sigma$

$WSS_i \equiv$ total demand frames • Approximation of locality

Working-Set Model (Cont.)

- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - Approximation of locality
- m = total number of frames
- If $D > m \Rightarrow$ Thrashing
- Policy if D > m, then suspend or swap out one of the processes

Keeping Track of theWorkingSet

- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page *i* – $B1_i$ and $B2_i$
  - Whenever a timer interrupts copy the reference toone

of the B_j and sets the values of all reference bits to0• If either $B1_i$ or $B2_i$ = 1, it implies that Page i is in the working set ▪ Why is this not completely accurate?

▪ Improvement = 10 bits and interrupt every 1000 time units

Working Sets andPageFaultRates

- Direct relationship between working set of a process anditspage-fault rate
- Working set changes over time
- Peaks and valleys over time

Page-Fault FrequencyAlgorithm

More direct approach than WSS

- Establish "acceptable"**page-fault frequency** (**PFF**) rateand use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high,

process gains frame

Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel

memory needs to be contiguous 4 i.e., for device I/O

- Two schemes:
  - Buddy System
  - Slab Allocator

Buddy System

- Allocates memory from fixed-size segment consistingof physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into

two buddies of next-lower power of 2

4 Continue until appropriate sized chunk available

Buddy SystemExample

- Assume 256KB chunk available, kernel requests 21KB
  - Split into A_L and A_R of 128KB each
    - 4 One further divided into B_L and B_R of 64KB
      - – One further into C_L and C_R of 32KB each –one used to satisfy request
- Advantage – quickly **coalesce** unused chunks into larger chunk
- Disadvantage - fragmentation

Buddy SystemAllocator

Slab Allocator ▪ Alternate strategy

▪ **Slab** is one or more physically contiguous pages ▪ **Cache** consists of one or more slabs

▪ Single cache for each unique kernel data structure • Each cache filled with **objects** – instantiations of thedatastructure

▪ When cache created, filled with objects marked as `free`▪ When structures stored, objects marked as `used` ▪ If slab is full of used objects, next object allocated fromemptyslab• If no empty slabs, new slab allocated

▪ Benefits include no fragmentation, fast memory request satisfaction

Slab Allocation

Slab Allocator inLinux

▪ For example, process descriptor is of type `struct task_struct`

▪ Approximately 1.7 KB of memory

▪ New task -> allocate new struct from cache

• Will use existing free `struct task_struct`

▪ Slab can be in three possible states

1. Full – all used
2. Empty – all free
3. Partial – mix of free and used

▪ Upon request, slab allocator

1. Uses free struct in partial slab
2. If none, takes one from empty slab
3. If no empty slab, create new empty

Slab Allocator inLinux(Cont.)

- Slab started in Solaris, now wide-spread for both kernel mode and user memory in various OSes
- Linux 2.2 had SLAB, now has both SLOB and SLUBallocators
  - SLOB for systems with limited memory
    - Simple List of Blocks – maintains 3 list objects for small, medium, large objects
  - SLUB is performance-optimized SLAB removes per- CPU queues, metadata stored in page structure

Other Considerations

- Prepaging
- Page size
- TLB reach
- Inverted page table
- Program structure
- I/O interlock and page locking

Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory waswasted
- Assume *s* pages are prepaged and *α* of the pages is used• Question: is the cost of ***s * α*** save pages faults is greater or less than the cost of prepaging ***s * (1- α)*** unnecessary pages?
  - If ***α*** is close to 0 $\Rightarrow$ prepaging loses
  - If ***α*** is close to 1 $\Rightarrow$ prepaging wins

Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - **Resolution**
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time

TLBReach

- TLB Reach - The amount of memory accessible fromtheTLB▪ TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in theTLB• Otherwise there is a high degree of page faults ▪ Increase the Page Size
  - This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes
  - This allows applications that require larger page sizestheopportunity to use them without an increase in fragmentation

ProgramStructure

- Program structure
  - `int[128,128] data;`
  - Each row is stored in one page
  - Program 1

```
 for (j = 0; j <128; j++)
for (i = 0; i < 128; i++)
                 data[i,j] = 0;
```

128 x 128 = 16,384 page faults

  - Program 2

```
for (i = 0; i < 128; i++)
       for (j = 0; j < 128; j++)
                 data[i,j] = 0;
```

128 page faults

I/Ointerlock

- **I/O Interlock** – Pages must sometimes be locked into memory
  - Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm
- **Pinning** of pages to lock into memory

Operating SystemExamples

- Windows
- Solaris

Windows

- Uses demand paging with **clustering**. Clustering brings inpagessurrounding the faulting page
- Processes are assigned **working set minimum**and **workingsetmaximum**

- Working set minimum is the minimum number of pages theprocess is guaranteed to have in memory
- A process may be assigned as many pages up to its workingset maximum
- When the amount of free memory in the systemfalls belowathreshold, **automatic working set trimming** is performedtorestore the amount of free memory
- Working set trimming removes pages fromprocesses that havepages in excess of their working set minimum

Solaris

- Maintains a list of free pages to assign faulting processes
- **`Lotsfree`** – threshold parameter (amount of free memory) tobegin paging
- **`Desfree`** – threshold parameter to increasing paging
- **`Minfree`** – threshold

parameter to being swapping ▪ Paging is performed by **`pageout`** process ▪ **`Pageout`** scans pages using modified clock algorithm▪ **`Scanrate`** is the rate at which pages are scanned. This rangesfrom **`slowscan`** to **`fastscan`**

- **`Pageout`** is called more frequently depending upon theamount of free memory available
  - ***Priority paging*** gives priority to process code pages

Solaris 2 PageScanner