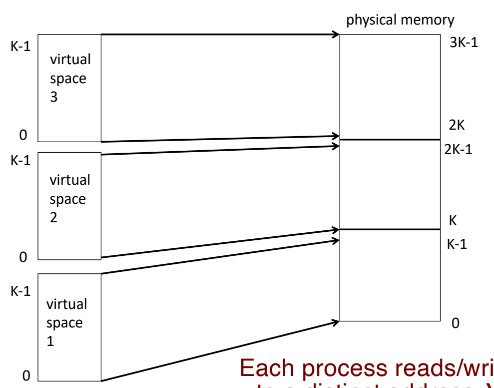
Operating Systems

Virtual Memory

Virtual Memory

- Separation of address space into physical and virtual addresses
  - Real vs. logical
- Implications
  - Processes can be isolated from one another
  - Virtual address space can be much larger than physical address space
  - Allows address spaces to be shared by processes
  - Only part of the program needs to be in memory for execution – the rest can be stored on disk
  - Allows for more efficient process creation

Process Isolation



Each process reads/write to a distinct address *X*

Managing the Mappings

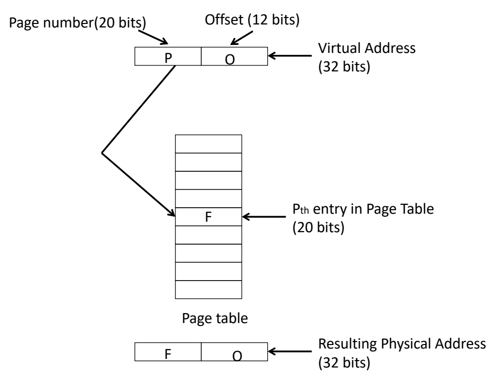
- The relocation register approach requires contiguous allocation of storage
  - One offset per process
- *Segmentation* is an approach in which a program's address space is separated into related chunks
  - Each chunk mapped separately, but contiguous within the chunk
  - Multiple segments per process
- The most flexible approach uses many small chunks and is known as *paging*

Paging

- Physical memory is divided into blocks called *frames*
  - Size is often 4K or 8K
  - large frames of 2/4MB on x86(_64), 16K on ARM (32 bit)
- Virtual memory *pages* are mapped into physical frames
- The size of a frame governs how many bits of the address are used represent the displacement inside a page, and how many represent a page/frame number

Page Tables

- Table of page->frame mappings stored in memory
- A register points to the base of the page table
- The MMU in the CPU uses the page number as an offset into the table
- The page table entry contains the frame number



Translation Lookaside Buffer (TLB)

- With no hardware support, each memory access would require an additional access to the page table
- The TLB is a cache for page table entries
  - typical sizes range from 64 to 4K entries
- Content addressable memory
  - Search all entries at once, constant time retrieval
- If present, fetch physical address and append page offset
- Else, go to the page table and get the frame address, updating the TLB
- Some processors feature multi-level TLBs

TLB and Context Switching

- Logical addresses in different processes have different physical addresses
- Flush the TLB?
- Some TLBs store an address-space identifier (ASID) in each TLB entry
  – Can uniquely identify a process to provide address-space protection for that process
  – Store the PID/TID in each TLB entry

Page Table Structure

- How should the page table be organized?
  – 2^20 PTEs = 4MB page tables per process
  – Far worse for 64-bit machines (even with 8K pages)
- One solution is Hierarchical Page tables

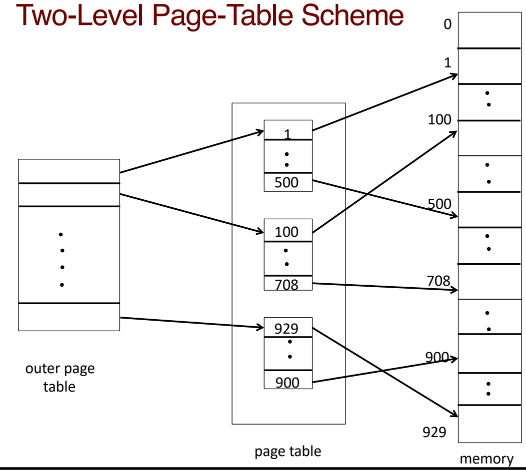
Two-Level Paging Example

- A logical address (32-bit address with 4K page size) is divided into:
  – a page number consisting of 20 bits
  – a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  – a 10-bit page directory number
  – A 10-bit page entry number
  – a 12-bit page offset
- Thus, a logical address looks like this:

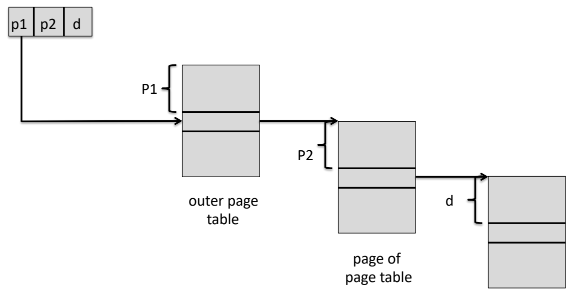
page number		page offset
p_i	p_2	d
10	10	12

where p_i is an index into the outer page table, and p_2 is the displacement within the page of the inner page table

Two-Level Page-Table Scheme

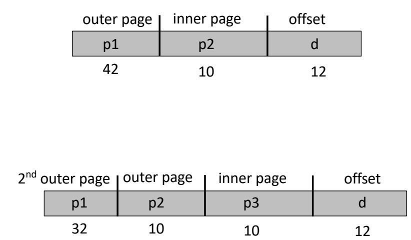


Address-Translation Scheme



13

Three-level Paging Scheme



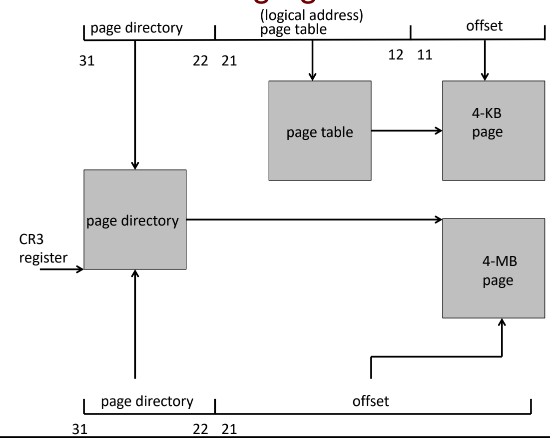
14

Example: Intel x86 family

- Supports segmentation with paging
  - Segmentation support removed in x86_64
- CPU generates logical address
  - Given to segmentation unit
    - Which produces linear addresses
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
- 2 partitions of segments
  - Private (local descriptor table)
  - Shared (global descriptor table)
- Segmentation could used to control execution from stack or heap memory

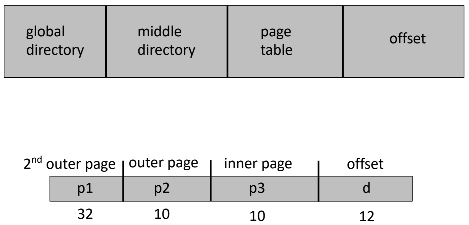
15

x86 Paging Architecture



16

Linear Addresses in Linux with three-level mapping



Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
- A page is needed when there is a reference to it
- If the page is not in memory, bring it in and resume

Memory Metadata

- The PTEs contain metadata bits about each page
  - Valid – is this page valid in this address space?
  - Present – is this page currently mapped to a physical memory frame?
  - Read Only
  - Executable
  - Modified (dirty)

Page Table Walks and Page Faults

- If there is a TLB miss, the MMU will walk the page table and load the appropriate PTE into the TLB
- If the PTE indicates that the page is not valid or present, the system raises a *page fault* exception
- The OS must examine the state of the page and take appropriate action
  - Valid but not in memory – bring page in and update the page table
  - Invalid – deliver SIGSEGV
  - Read-only – make copy or deliver error

Shared Code

- Shared code must be either
  - Position Independent Code (PIC)
  - Mapped in the same location in all processes
- PIC does not contain absolute addresses
- PC-relative
  - Reference data as an offset from the program counter
  - Or relative to a PIC register
- Indirect addressing
  - Linkage Table or Global Offset Table stores addresses after dynamic linking
- Windows DLLs try to be mapped at different fixed addresses

Copy-on-Write

- Copy-on-Write allows parent and child processes to initially share pages
- The page is marked read-only
- If either process attempts to write to a shared page, it generates an exception
- Trap to the kernel, make a copy of the page, update both processes' page table, resume
- COW makes process creation faster and can use less memory when both parent and child continue to execute the same program

Page Replacement

- When there are no free frames, some page must be selected for eviction
- The algorithm for choosing which page should be chosen to minimize future page faults
- Use **modified** metadata indicator to reduce overhead of page transfers – only modified pages are written to disk
- Find a free frame:
  - If there is a free frame, use it
  - If there is no free frame, use a page replacement algorithm to select a victim frame
- Bring the desired page into the (newly) free frame; update the page tables
- Restart the process

Replacing Pages

- Eventually, all free frames will be consumed
- Some pages must be selected for eviction
- The goal is to minimize future page faults
- Generally, systems try to run page eviction in the background, keeping some free frames on hand

First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

- 4 frames

1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		

- Belady's Anomaly: more frames ⇒ more page faults

Optimal Algorithm

- Replace page that will not be used for longest period of time
- 4 frames example

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	4	
2		6 page faults
3		
4	5	

- How would one know this? You can't.
- Useful for evaluating how well a given algorithm performs

Least Recently Used (LRU) Algorithm

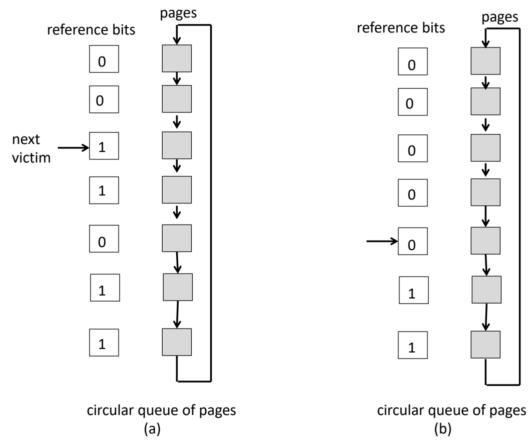
- Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**

1	1	1	1	**5**
2	2	2	2	2
3	**5**	5	**4**	4
4	4	**3**	3	3

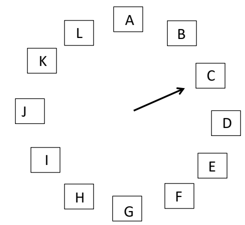
- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to determine which are to change

LRU Approximation Algorithms

- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced, the reference bit is set to 1
  - Replace the one which is 0 (if one exists)
    - We do not know the order, however
- Second chance
  - Need reference bit
  - Clock replacement
  - If page to be replaced (in clock order) has reference bit = 1 then:
    - set reference bit 0
    - leave page in memory
    - replace next page (in clock order), subject to same rules



The Clock Page Replacement Algorithm



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R= 0: Evict the page
R= 1: Clear R and advance hand

Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the 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

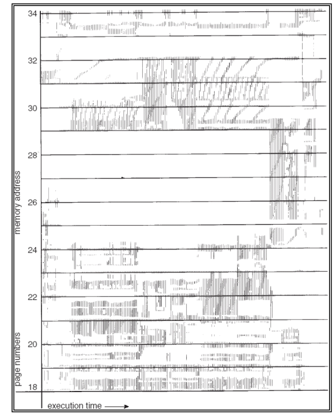
Thrashing

- If a process does not have "enough" pages, the page-fault rate will be high. This can lead to:
  – low CPU utilization
  – operating system thinks that it needs to increase the degree of multiprogramming
  – another process added to the system
- The system is spending more time handling page faults than doing actual work

Demand Paging and Thrashing

- 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

Locality In A Memory-Reference Pattern



Locality of Reference

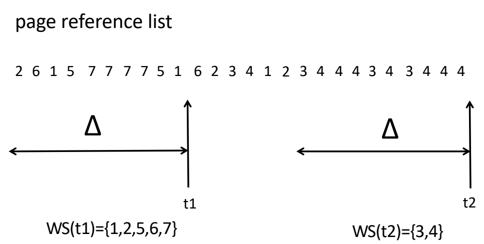
- Spatial locality
  – Likely to access nearby locations in memory
  – Sequential array access
  – Fields in a structure
- Temporal locality
  – Multiple accesses to a variable happen close in time

```
for ( i=0; i<10; i++ ) { …
```


Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instruction
- WSS_i (working set of Process P_i) =
  total number 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
- $D = \Sigma\, WSS_i \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing
- Policy if $D >$ m, then suspend one of the processes

Working-set model



Keeping Track of the Working Set

Approximate with interval timer + a reference bit

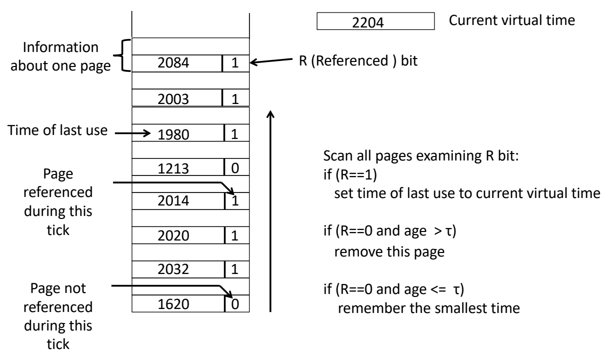
Example: Δ = 10,000

- Timer interrupts after every 5000 time units
- Keep 2 bits in memory for each page
- Whenever a timer interrupts, copy and set the values of all reference bits to 0
- If one of the bits in memory = 1 $\Rightarrow$ page in working set

Why is this not completely accurate?

Potential improvement = 10 bits and interrupt every 1000 time units

Working set page replacement



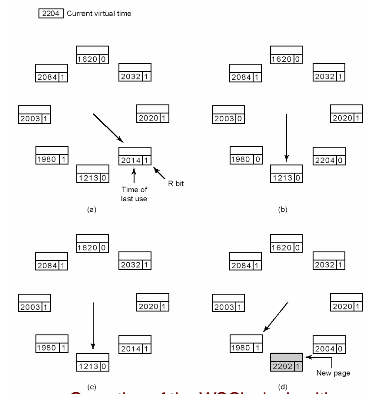
WSClock page replacement algorithm

- An implementation of the working set algorithm
- All pages are kept in a circular list (ring)
- As pages are added, they go into the ring
- The "clock hand" advances around the ring
- Each entry contains "time of last use"
- Upon a page fault...
  - If Reference Bit = 1...
    - Page is in use now. Do not evict.
    - Clear the Referenced Bit
    - Update the "time of last use" field

WSClock page replacement algorithm

- If Reference Bit = 0
  - If the age of the page is less than T...
    - This page is in the working set.
    - Advance the hand and keep looking
  - If the age of the page is greater than T...
    - If page is clean
      - Reclaim the frame and we are done
    - If page is dirty
      - Schedule a write for the page
      - Advance the hand and keep looking

The WSClock Page Replacement Algorithm



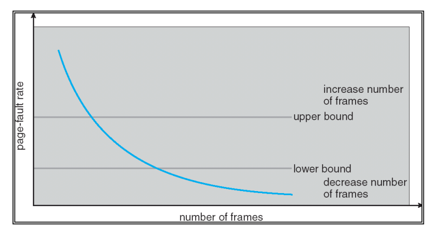
Operation of the WSClock algorithm

WS-Clock

- What happens when R=0?
  - Is age > T, and page is clean then it is evicted
    - If it is dirty then we can proceed to find a page that may be clean and avoid a process switch
    - We will still need to write to disk and this write is scheduled
- What happens if we go all the way around? (Two Scenarios)
  - At least one write has been scheduled. Keep looking… one will eventually be written to disk
  - No writes have been scheduled… all pages are in the working set, so evict a clean page or the current page if a clean page does not exist

Page-Fault Frequency Scheme

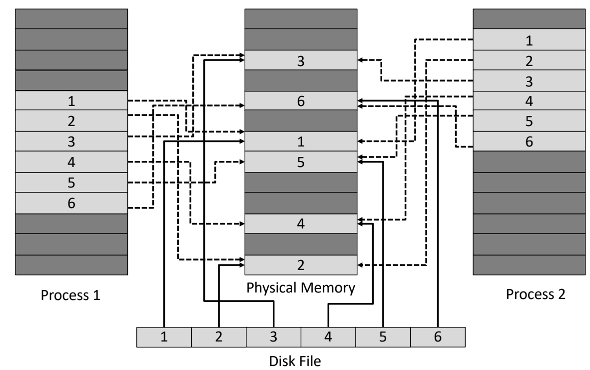
- Establish “acceptable” page-fault rate
  - If actual rate too low, process loses frames
  - If actual rate too high, process gains frames



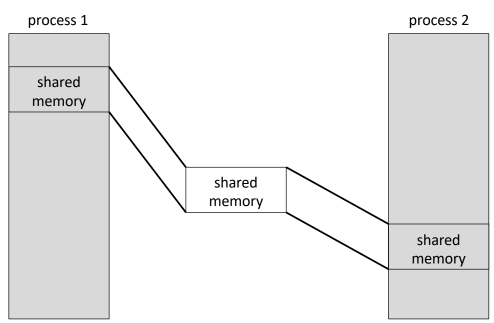
Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- Simplifies file access by treating file I/O through memory rather than `read()` `write()` system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared

Memory Mapped Files



Memory-Mapped Shared Memory



Page Size

- Page size selection must take into consideration:
  - fragmentation
  - table size
  - I/O overhead
  - locality

TLB Reach

- TLB Reach - The amount of memory able to be referenced from the TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in the TLB
  - Otherwise, there are TLB misses
- 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 sizes the opportunity to use them without an increase in fragmentation

49

Program Structure

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

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

$\rightarrow$ 128 x 128 = 16,384 page faults

  - Version 2

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

$\rightarrow$ 128 page faults

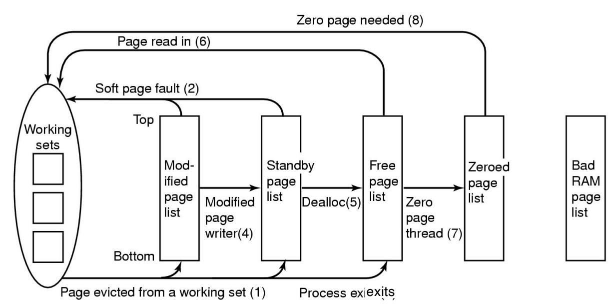
50

Other Issues -- Prepaging

- 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 was wasted
  - Assume *s* pages are prepaged and α of the pages are used
    - Is cost of $s * \alpha$ saved pages faults > or < than the cost of prepaging $s * (1- \alpha)$ unnecessary pages?
    - α near zero $\Rightarrow$ prepaging loses

51

Windows Memory Management



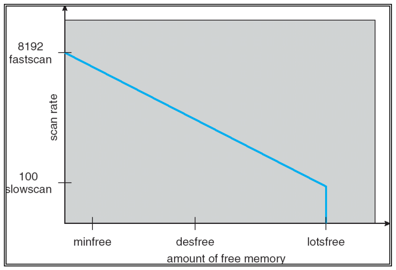
The various page lists and the transitions between them

52

Solaris

- Maintains a list of free pages to assign faulting processes
- *Lotsfree* – threshold parameter (amount of free memory) to begin 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 ranges from *slowscan* to *fastscan*
- Pageout is called more frequently depending upon the amount of free memory available

Solaris 2 Page Scanner



Algorithm Summary

Algorithm	Comment
Optimal	Not implementable, but useful as benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second Chance	Big improvement over FIFO
Clock	Realistic
LRU(Least Recently Used)	Excellent, but difficult to implement exactly
NFU(Not frequently used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSCLock	Good, efficient algorithm