CS194-24
Advanced Operating Systems Structures and Implementation
Lecture 12

Paging
Memory Allocation and
File Systems

March 5th, 2014

Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs194-24

Goals for Today

- Paging (Con't)
- Memory Allocation
- File Systems

Interactive is important!
Ask Questions!

Note: Some slides and/or pictures in the following are adapted from slides ©2013

Recall: What is in a Page Table Entry (PTE)?

- What is in a Page Table Entry (or PTE)?
  - Pointer to next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called "Directories"

P: Present (same as "valid" bit in other architectures)
W: Writeable
U: User accessible
PWT: Page write transparent: external cache write-through
PCD: Page cache disabled (page cannot be cached)
A: Accessed: page has been accessed recently
D: Dirty (PTE only): page has been modified recently
L: L=1⇒4MB page (directory only).
Bottom 22 bits of virtual address serve as offset

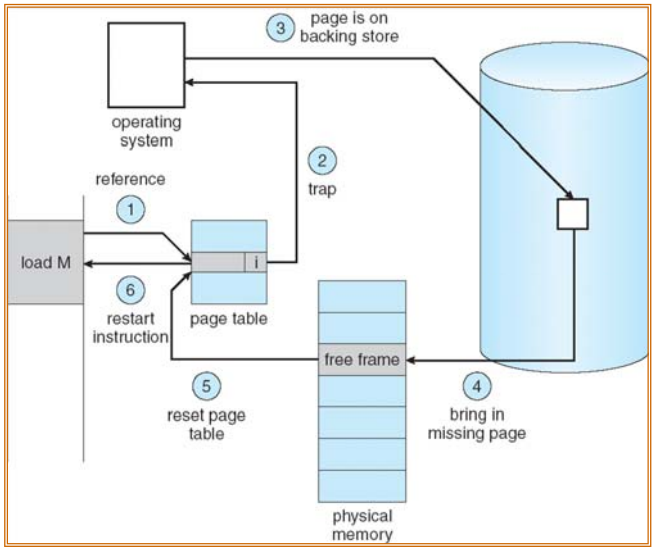
Recall: Ways to exploit the PTE

- How do we use the PTE?
  - Invalid PTE can imply different things:
    - » Region of address space is actually invalid or
    - » Page/directory is just somewhere else than memory
  - Validity checked first
    - » OS can use other (say) 31 bits for location info
- Usage Example: Demand Paging
  - Keep only active pages in memory
  - Place others on disk and mark their PTEs invalid
- Usage Example: Copy on Write
  - UNIX fork gives *copy* of parent address space to child
    - » Address spaces disconnected after child created
  - How to do this cheaply?
    - » Make copy of parent's page tables (point at same memory)
    - » Mark entries in both sets of page tables as read-only
    - » Page fault on write creates two copies
- Usage Example: Zero Fill On Demand
  - New data pages must carry no information (say be zeroed)
  - Mark PTEs as invalid; page fault on use gets zeroed page
  - Often, OS creates zeroed pages in background

Recall: Demand Paging Mechanisms

- PTE helps us implement demand paging
  - Valid ⇒ Page in memory, PTE points at physical page
  - Not Valid ⇒ 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

Steps in Handling a Page Fault

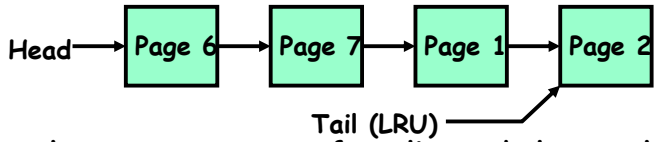


Page Replacement Policies

- 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

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!

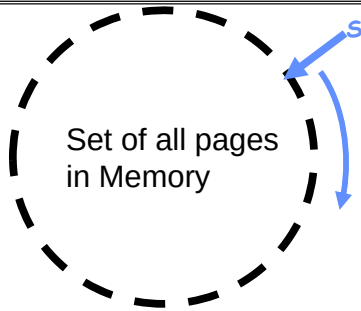


  - On each use, remove page from list and place at head
  - LRU page is at tail
- Problems with this scheme for paging?
  - Need to know immediately when each page used so that can change position in list…
  - Many instructions for each hardware access
- In practice, people approximate LRU (more later)

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

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?

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)

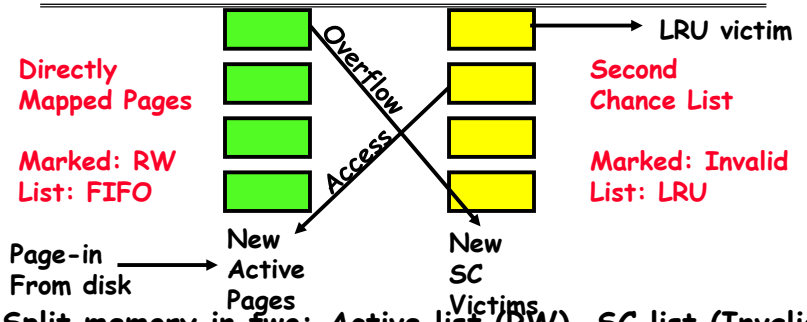
Clock Algorithms: Details

- 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 software "modified" bit, and marks page as read-write.
    - » Whenever page comes back in from disk, mark read-only

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

Second-Chance List Algorithm (VAX/VMS)

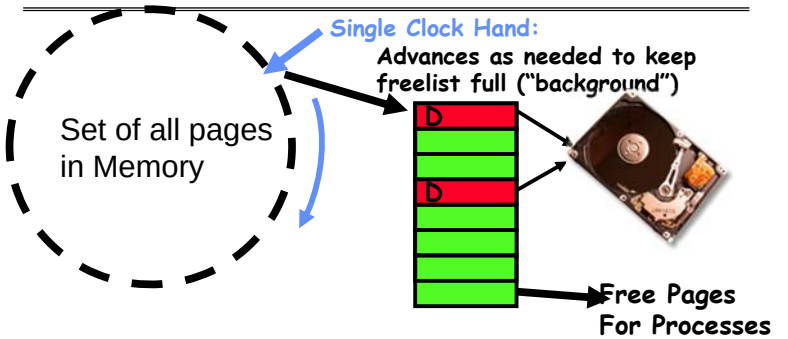


- Split memory in two: Active list (RW), SC list (Invalid)
- Access pages in Active list at full speed
- Otherwise, Page Fault
  - Always move overflow page from end of Active list to front of Second-chance list (SC) and mark invalid
  - Desired Page On SC List: move to front of Active list, mark RW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

Second-Chance List Algorithm (con't)

- How many pages for second chance list?
  - If 0 ⇒ FIFO
  - If all ⇒ 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

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

Reverse Page Mapping (Sometimes called "Coremap")

- Physical page frames often shared by many different address spaces/page tables
  - All children forked from given process
  - Shared memory pages between processes
- Whatever reverse mapping mechanism that is in place must be very fast
  - Must hunt down all page tables pointing at given page frame when freeing a page
  - Must hunt down all PTEs when seeing if pages "active"
- Implementation options:
  - For every page descriptor, keep linked list of page table entries that point to it
    - » Management nightmare – expensive
  - Linux 2.6: Object-based reverse mapping
    - » Link together memory region descriptors instead (much coarser granularity)

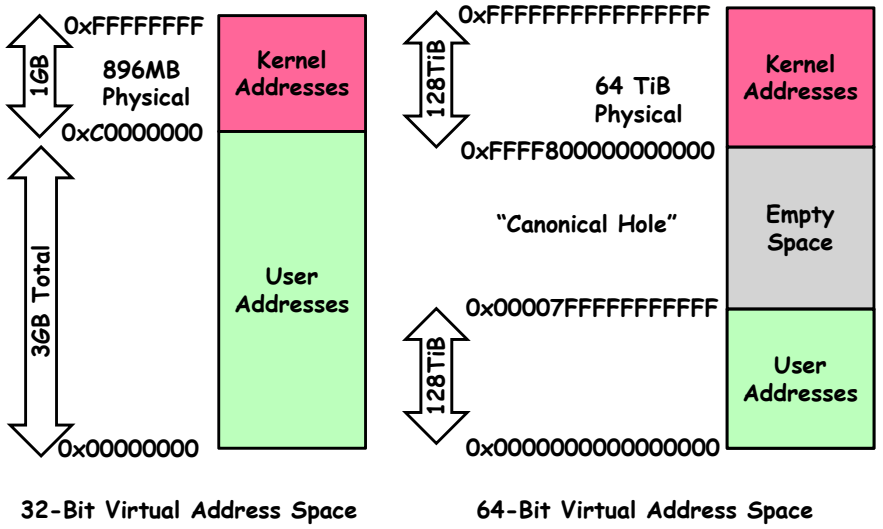
Administrivia

- Midterm I: Next Wednesday (3/12)!
  - Intention is a 1.5 hour exam over 3 hours
  - No class on day of exam!
- Midterm Timing:
  - 7:00-10:00PM in 306 Soda Hall (Here!)
- Topics: everything up to Monday
  - OS Structure, BDD, Process support, Synchronization, Memory Management, File systems
  - Labs/Papers

Linux Memory Details?

- Memory management in Linux considerably more complex that the previous indications
- Memory Zones: physical memory categories
  - ZONE_DMA: < 16MB memory, DMAable on ISA bus
  - ZONE_NORMAL: 16MB ⇒ 896MB (mapped at 0xC0000000)
  - ZONE_HIGHMEM: Everything else (> 896MB)
- Each zone has 1 freelist, 2 LRU lists (Active/Inactive)
- Many different types of allocation
  - SLAB allocators, per-page allocators, mapped/unmapped
- Many different types of allocated memory:
  - Anonymous memory (not backed by a file, heap/stack)
  - Mapped memory (backed by a file)
- Allocation priorities
  - Is blocking allowed/etc

Recall: Linux Virtual memory map



Virtual Map (Details)

- Kernel memory not generally visible to user
  - Exception: special VDSO facility that maps kernel code into user space to aid in system calls (and to provide certain actual system calls such as gettimeofday().
- Every physical page described by a "page" structure
  - Collected together in lower physical memory
  - Can be accessed in kernel virtual space
  - Linked together in various "LRU" lists
- For 32-bit virtual memory architectures:
  - When physical memory < 896MB
    - » All physical memory mapped at 0xC0000000
  - When physical memory >= 896MB
    - » Not all physical memory mapped in kernel space all the time
    - » Can be temporarily mapped with addresses > 0xCC000000
- For 64-bit virtual memory architectures:
  - All physical memory mapped above 0xFFFF800000000000

Internal Interfaces: Allocating Memory

- One mechanism for requesting pages: everything else on top of this mechanism:
  - Allocate contiguous group of pages of size 2^{order} bytes given the specified mask:

```
struct page * alloc_pages(gfp_t gfp_mask,
                          unsigned int order)
```

  - Allocate one page:

```
struct page * alloc_page(gfp_t gfp_mask)
```

  - Convert page to logical address (assuming mapped):

```
void * page_address(struct page *page)
```

- Also routines for freeing pages
- Zone allocator uses "buddy" allocator that trys to keep memory unfragmented
- Allocation routines pick from proper zone, given flags

Allocation flags

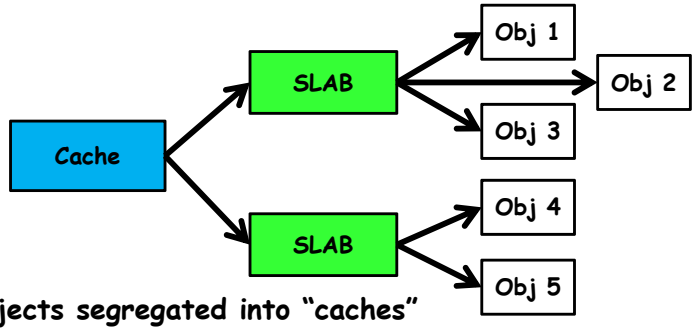
- Possible allocation type flags:
  - GFP_ATOMIC: Allocation high-priority and must never sleep. Use in interrupt handlers, top halves, while holding locks, or other times cannot sleep
  - GFP_NOWAIT: Like GFP_ATOMIC, except call will not fall back on emergency memory pools. Increases likely hood of failure
  - GFP_NOIO: Allocation can block but must not initiate disk I/O.
  - GFP_NOFS: Can block, and can initiate disk I/O, but will not initiate filesystem ops.
  - GFP_KERNEL: Normal allocation, might block. Use in process context when safe to sleep. This should be default choice
  - GFP_USER: Normal allocation for processes
  - GFP_HIGHMEM: Allocation from ZONE_HIGHMEM
  - GFP_DMA Allocation from ZONE_DMA. Use in combination with a previous flag

Page Frame Reclaiming Algorithm (PFRA)

- Several entrypoints:
  - Low on Memory Reclaiming: The kernel detects a "low on memory" condition
  - Hibernation reclaiming: The kernel must free memory because it is entering in the suspend-to-disk state
  - Periodic reclaiming: A kernel thread is activated periodically to perform memory reclaiming, if necessary
- Low on Memory reclaiming:
  - Start flushing out dirty pages to disk
  - Start looping over all memory nodes in the system
    - » try_to_free_pages()
    - » shrink_slab()
    - » pdflush kenel thread writing out dirty pages
- Periodic reclaiming:
  - Kswapd kernel threads: checks if number of free page frames in some zone has fallen below pages_high watermark
  - Each zone keeps two LRU lists: Active and Inactive
    - » Each page has a last-chance algorithm with 2 count
    - » Active page lists moved to inactive list when they have been idle for two cycles through the list
    - » Pages reclaimed from Inactive list

SLAB Allocator

- Replacement for free-lists that are hand-coded by users
  - Consolidation of all of this code under kernel control
  - Efficient when objects allocated and freed frequently



- Objects segregated into "caches"
  - Each cache stores different type of object
  - Data inside cache divided into "slabs", which are continuous groups of pages (often only 1 page)
  - Key idea: avoid memory fragmentation

SLAB Allocator Details

- Based on algorithm first introduced for SunOS
  - Observation: amount of time required to initialize a regular object in the kernel exceeds the amount of time required to allocate and deallocate it
  - Resolves around object caching
    - » Allocate once, keep reusing objects
- Avoids memory fragmentation:
  - Caching of similarly sized objects, avoid fragmentation
  - Similar to custom freelist per object
- Reuse of allocation
  - When new object first allocated, constructor runs
  - On subsequent free/reallocation, constructor does not need to be reexecuted

SLAB Allocator: Cache Construction

- Creation of new Caches:

```
struct kmem_cache * kem_cache_create(const char *name,
                                     size_t size,
                                     size_t align,
                                     unsigned long flags,
                                     void (*ctor)(void *));
```

  - name: name of cache
  - size: size of each element in the cache
  - align: alignment for each object (often 0)
  - flags: possible flags about allocation
    - » SLAB_HWCACHE_ALIGN: Align objects to cache lines
    - » SLAB_POISON: Fill slabs to known value (0xa5a5a5a5) in order to catch use of uninitialized memory
    - » SLAB_RED_ZONE: Insert empty zones around objects to help detect buffer overruns
    - » SLAB_PANIC: Allocation layer panics if allocation fails
    - » SLAB_CACHE_DMA: Allocations from DMA-able memory
    - » SLAB_NOTRACK: don't track uninitialized memory
  - ctor: called whenever new pages are added to the cache

SLAB Allocator: Cache Use

- Example:

```
task_struct_cachep =
    kmem_cache_create("task_struct",
                      sizeof(struct task_struct),
                      ARCH_MIN_TASKALIGN,
                      SLAB_PANIC | SLAB_NOTRACK,
                      NULL);
```

- Use of example:

```
struct task_struct *tsk;

tsk = kmem_cache_alloc(task_struct_cachep, GFP_KERNEL);
if (!tsk)
    return NULL;

kmem_free(task_struct_cachep,tsk);
```

SLAB Allocator Details (Con't)

- Caches can be later destroyed with:
  `int kmem_cache_destroy(struct kmem_cache *cachep);`
  - Assuming that all objects freed
  - No one ever tries to use cache again
- All caches kept in global list
  - Including global caches set up with objects of powers of 2 from 2^5 to 2^{17}
  - General kernel allocation (kmalloc/kfree) uses least-fit for requested cache size
- Reclamation of memory
  - Caches keep sorted list of empty, partial, and full slabs
    - » Easy to manage – slab metadata contains reference count
    - » Objects within slabs linked together
  - Ask individual caches for full slabs for reclamation

Recall: Kmalloc/Kfree: The "easy interface" to memory

- Simplest kernel interface to manage memory: `kmalloc()/kfree()`
  - Allocate chunk of memory in kernel's address space (will be physically contiguous and virtually contiguous):

```
void * kmalloc(size_t size, gfp_t flags);
```

  - Example usage:

```
struct dog *p;
p = kmalloc(sizeof(struct dog), GFP_KERNEL);
if (!p)
        /* Handle error! */
```

  - Free memory: `void kfree(const void *ptr);`
  - Important restrictions!
    - » Must call with memory previously allocated through `kmalloc()` interface!!!
    - » Must not free memory twice!

Alternatives for allocation

- According to Robert Love, "SLAB" has become a name for any allocator with a similar API
  - Kinda like "Kleenex" has become a generic noun
- A number of options in the kernel for object allocation:
  - SLAB: original allocator based on Bonwick's paper from SunOS
  - SLUB: Newer allocator with same interface but better use of metadata (Default since Linux 2.6.23)
    - » Keeps SLAB metadata in the page data structure (for pages that happen to be in kernel caches)
    - » Debugging options compiled in by default, just need to be enabled
  - SLOB: low-memory footprint allocator for embedded systems

Designing the File System: Access Patterns

- How do users access files?
  - Need to know type of access patterns user is likely to throw at system
- Sequential Access: bytes read in order ("give me the next X bytes, then give me next, etc")
  - Almost all file access are of this flavor
- Random Access: read/write element out of middle of array ("give me bytes i—j")
  - Less frequent, but still important. For example, virtual memory backing file: page of memory stored in file
  - Want this to be fast – don't want to have to read all bytes to get to the middle of the file
- Content-based Access: ("find me 100 bytes starting with KUBIATOWICZ")
  - Example: employee records – once you find the bytes, increase my salary by a factor of 2
  - Many systems don't provide this; instead, databases are built on top of disk access to index content (requires efficient random access)

Designing the File System: Usage Patterns

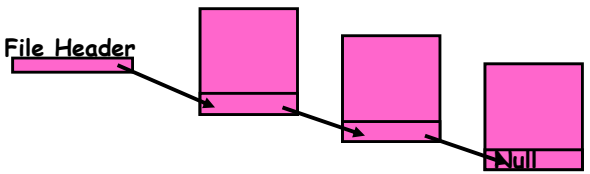
- Most files are small (for example, .login, .c files)
  - A few files are big – nachos, core files, etc.; the nachos executable is as big as all of your .class files combined
  - However, most files are small – .class's, .o's, .c's, etc.
- Large files use up most of the disk space and bandwidth to/from disk
  - May seem contradictory, but a few enormous files are equivalent to an immense # of small files
- Although we will use these observations, beware usage patterns:
  - Good idea to look at usage patterns: beat competitors by optimizing for frequent patterns
  - Except: changes in performance or cost can alter usage patterns. Maybe UNIX has lots of small files because big files are really inefficient?

How to organize files on disk

- Goals:
  - Maximize sequential performance
  - Easy random access to file
  - Easy management of file (growth, truncation, etc)
- First Technique: Continuous Allocation
  - Use continuous range of blocks in logical block space
    - » Analogous to base+bounds in virtual memory
    - » User says in advance how big file will be (disadvantage)
  - Search bit-map for space using best fit/first fit
    - » What if not enough contiguous space for new file?
  - File Header Contains:
    - » First sector/LBA in file
    - » File size (# of sectors)
  - Pros: Fast Sequential Access, Easy Random access
  - Cons: External Fragmentation/Hard to grow files
    - » Free holes get smaller and smaller
    - » Could compact space, but that would be *really* expensive
- Continuous Allocation used by IBM 360
  - Result of allocation and management cost: People would create a big file, put their file in the middle

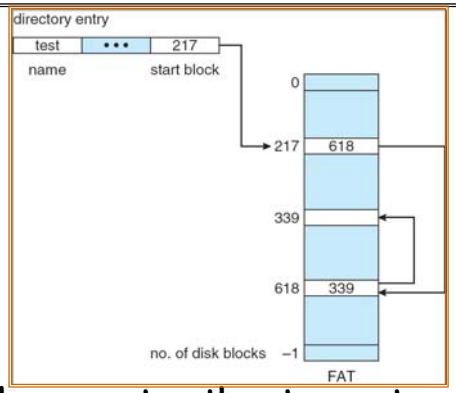
Linked List Allocation

- Second Technique: Linked List Approach
  - Each block, pointer to next on disk



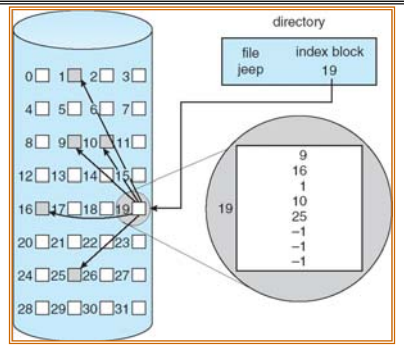
  - Pros: Can grow files dynamically, Free list same as file
  - Cons: Bad Sequential Access (seek between each block), Unreliable (lose block, lose rest of file)
  - Serious Con: Bad random access!!!!
  - Technique originally from Alto (First PC, built at Xerox)
    - » No attempt to allocate contiguous blocks

Linked Allocation: File-Allocation Table (FAT)



- MSDOS links pages together to create a file
  - Links not in pages, but in the File Allocation Table (FAT)
    - » FAT contains an entry for each block on the disk
    - » FAT Entries corresponding to blocks of file linked together
  - Access properies:
    - » Sequential access expensive unless FAT cached in memory
    - » Random access expensive always, but *really* expensive if FAT not cached in memory

Indexed Allocation



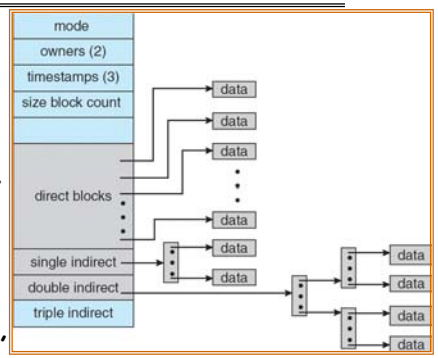
- Indexed Files (Nachos, VMS)
  - System Allocates file header block to hold array of pointers big enough to point to all blocks
    - » User pre-declares max file size;
  - Pros: Can easily grow up to space allocated for index
    Random access is fast
  - Cons: Clumsy to grow file bigger than table size
    Still lots of seeks: blocks may be spread over disk

Multilevel Indexed Files (UNIX BSD 4.1)

- Multilevel Indexed Files: Like multilevel address translation (from UNIX 4.1 BSD)
  - Key idea: efficient for small files, but still allow big files
  - File header contains 13 pointers
    - » Fixed size table, pointers not all equivalent
    - » This header is called an "inode" in UNIX
  - File Header format:
    - » First 10 pointers are to data blocks
    - » Block 11 points to "indirect block" containing 256 blocks
    - » Block 12 points to "doubly indirect block" containing 256 indirect blocks for total of 64K blocks
    - » Block 13 points to a triply indirect block (16M blocks)
- Discussion
  - Basic technique places an upper limit on file size that is approximately 16Gbytes
    - » Designers thought this was bigger than anything anyone would need. Much bigger than a disk at the time…
    - » Fallacy: today, EOS producing 2TB of data per day
  - Pointers get filled in dynamically: need to allocate indirect block only when file grows > 10 blocks.
    - » On small files, no indirection needed

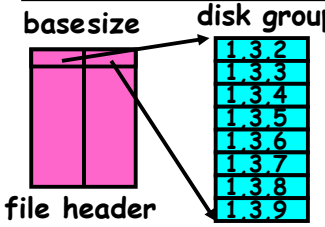
Example of Multilevel Indexed Files

- Sample file in multilevel indexed format:
  - How many accesses for block #23? (assume file header accessed on open)?
    - » Two: One for indirect block, one for data
  - How about block #5?
    - » One: One for data
  - Block #340?
    - » Three: double indirect block, indirect block, and data



- UNIX 4.1 Pros and cons
  - Pros: Simple (more or less)
    Files can easily expand (up to a point)
    Small files particularly cheap and easy
  - Cons: Lots of seeks
    Very large files must read many indirect block (four I/Os per block!)

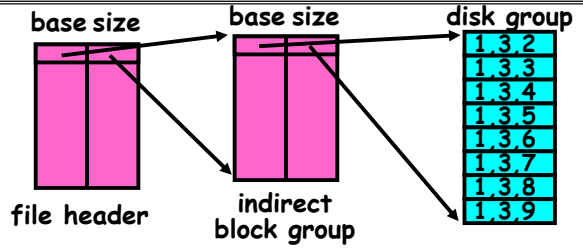
File Allocation for Cray-1 DEMOS



Basic Segmentation Structure:
Each segment contiguous on disk

- DEMOS: File system structure similar to segmentation
  - Idea: reduce disk seeks by
    - » using contiguous allocation in normal case
    - » but allow flexibility to have non-contiguous allocation
  - Cray-1 had 12ns cycle time, so CPU:disk speed ratio about the same as today (a few million instructions per seek)
- Header: table of base & size (10 "block group" pointers)
  - Each block chunk is a contiguous group of disk blocks
  - Sequential reads within a block chunk can proceed at high speed – similar to continuous allocation
- How do you find an available block group?
  - Use freelist bitmap to find block of 0's.

Large File Version of DEMOS



- What if need much bigger files?
  - If need more than 10 groups, set flag in header: BIGFILE
    - » Each table entry now points to an indirect block group
  - Suppose 1000 blocks in a block group ⇒ 80GB max file
    - » Assuming 8KB blocks, 8byte entries⇒ (10 ptrs×1024 groups/ptr×1000 blocks/group)*8K =80GB
- Discussion of DEMOS scheme
  - Pros: Fast sequential access, Free areas merge simply Easy to find free block groups (when disk not full)
  - Cons: Disk full ⇒ No long runs of blocks (fragmentation), so high overhead allocation/access
  - Full disk ⇒ worst of 4.1BSD (lots of seeks) with worst of continuous allocation (lots of recompaction needed)

How to keep DEMOS performing well?

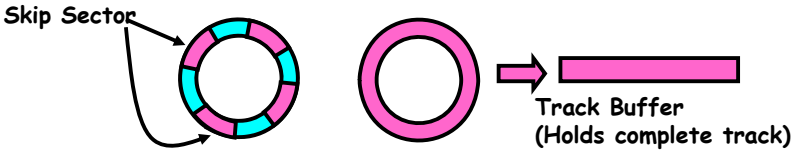
- In many systems, disks are always full
  - How to fix? Announce that disk space is getting low, so please delete files?
    - » Don't really work: people try to store their data faster
  - Sidebar: Perhaps we are getting out of this mode with new disks… However, let's assume disks full for now
- Solution:
  - Don't let disks get completely full: reserve portion
    - » Free count = # blocks free in bitmap
    - » Scheme: Don't allocate data if count < reserve
  - How much reserve do you need?
    - » In practice, 10% seems like enough
  - Tradeoff: pay for more disk, get contiguous allocation
    - » Since seeks so expensive for performance, this is a very good tradeoff

UNIX BSD 4.2

- Same as BSD 4.1 (same file header and triply indirect blocks), except incorporated ideas from DEMOS:
  - Uses bitmap allocation in place of freelist
  - Attempt to allocate files contiguously
  - 10% reserved disk space
  - Skip-sector positioning (mentioned next slide)
- Problem: When create a file, don't know how big it will become (in UNIX, most writes are by appending)
  - How much contiguous space do you allocate for a file?
  - In Demos, power of 2 growth: once it grows past 1MB, allocate 2MB, etc
  - In BSD 4.2, just find some range of free blocks
    - » Put each new file at the front of different range
    - » To expand a file, you first try successive blocks in bitmap, then choose new range of blocks
  - Also in BSD 4.2: store files from same directory near each other
- Fast File System (FFS)
  - Allocation and placement policies for BSD 4.2

Attack of the Rotational Delay

- Problem 2: Missing blocks due to rotational delay
  - Issue: Read one block, do processing, and read next block. In meantime, disk has continued turning: missed next block! Need 1 revolution/block!



  - Solution1: Skip sector positioning ("interleaving")
    - » Place the blocks from one file on every other block of a track: give time for processing to overlap rotation
  - Solution2: Read ahead: read next block right after first, even if application hasn't asked for it yet.
    - » This can be done either by OS (read ahead)
    - » By disk itself (track buffers). Many disk controllers have internal RAM that allows them to read a complete track
- Important Aside: Modern disks+controllers do many complex things "under the covers"
  - Track buffers, elevator algorithms, bad block filtering

Conclusion

- SLAB Allocator
  - Consolidation of allocation code under single API
  - Efficient when objects allocated and freed frequently
- Multilevel Indexed Scheme
  - Inode contains file info, direct pointers to blocks,
  - indirect blocks, doubly indirect, etc..
- Cray DEMOS: optimization for sequential access
  - Inode holds set of disk ranges, similar to segmentation
- 4.2 BSD Multilevel index files
  - Inode contains pointers to actual blocks, indirect blocks, double indirect blocks, etc
  - Optimizations for sequential access: start new files in open ranges of free blocks
  - Rotational Optimization