CMPUT379 Notes

Concurrency bugs

Non-deadlock bugs

- Atomicity bugs, order-violation bugs

Deadlock bugs

- Two threads access shared data, using different protocols
  - for example, one thread acquires lock 1, and another acquires lock 2.
  - thread 1 wants to acquire lock 2, but thread 2 wants to acquire lock 1 -> deadlock

Swapping

- If all processes do not fit in the memory, we acn preempt idle or mostly idle processes and reclaim their memory (sent to the backing store)
- When a process becomes active again, the OS swaps it back into memory using the metadata it has about that process.
  - with static reloaction, must be put in same position
  - with dynamic relocation, OS finds a new position in memory for the process and updates the relocation and limit registers
- Compaction support is easier with swapping

Segmentation

- Each process generates a contiguous virtual address from 0 to MAX_{proc}
- OS lays down the process on arbitrary size segments, and hardware translates virt addresses to phys in memory
  - this is done with a segment table that keeps track of locations
- virtual address identified by a pair - segment # and offset (s, d)
- This enables sharing code and data between processes (e.g. linking libraries)

When segments aren't allocated contiguously in memory for a process, each process gets a segment table that contains base and limit register values for each of the process segments.

The compiler generates references that identify the segment (code, global vars, stack, heap), and the offset in the segment (e.g. a code segment with offset 399)

Is segmentation enough?

- Leads to poor memory utilization
  - might not use much of a large segment, but have to keep the whole thing in memory
  - no standard size for segments, so allocation/deallocation of arbitrary sized segments is complex, and we get external fragmentation
- We can improve memory management by using fixed-size allocation units.

Paging

Motivation

- 90/10 rule: processes spend **90%** of their time accessing **10%** of their memory
  - keep the 10% that's being used in memory
- Logical memory of a process is contiguous, but pages don't need to be allocated contiguously in memory

- Paging eliminates external fragmentation and the associated need for fragmentation by dividing memory into fixed-size pages, e.g. 4096 bytes
- Paging does cause internal fragmentation - on average half a page is lost per process

Paging breaks physical memory into fixed size blocks, called **frames**, and breaks logical memory of each process into fixed size **pages**. We load pages of a process that is to be executed to the available page frames in memory.
- The per-process page table stores this mapping of page # → frame #
- To relocate each page dynamically, we just need to update the corresponding frame in the page table.

How to resolve addresses when pages aren't allocated contiguously in memory?
- virtual/logical address is divided into page number (p) and page offset (d). p is an index into the page table, extracting the frame number (f)
- The OS keeps track of free frames in the frame table which has an entry for each physical page frame (physical memory)—OS manages pages in memory

Paging details
- Page size usually a power of 2 (512 to 8192 bytes per page) → translation of virtual addresses into physical addresses is easy
- Given a memory size of 2^m bytes and a page/frame size of 2^n, then:
  - The high order $m - n$ bits of a virtual address select the page
  - the low order n bits are the offset in the page

For example, consider the memory size to be 2^{16} bytes, and we have 13 bits for the page offset.
- What is the page size? 2^{13} bytes
- How many frames do we have? 2^{16-13} = 8 frames
- Assuming we can address 1 byte increments, how many bits for an address? 16 bits - 3 bits for page table, 13 bits for offset
- Assuming we can address 1 word (2 bytes) increments, how many bits for an address? 15 bits (3 bits for page table, 12 bits for offset)

Calculating internal fragmentation
Reminder - external fragmentation is unused memory between units of allocation, and internal fragmentation is unused memory within a unit of allocation.
- worst case internal fragmentation is when the process needs just barely more than a multiple of the page size, e.g. the last block needs 1 byte.
- On average, internal fragmentation is 1/2 of a page size

If the page/frame size is 2048 bytes and process size is 72766 bytes, that's 35 pages plus 1086 bytes
- internal fragmentation of 2048 - 1086 = 962 bytes

But smaller frame sizes aren't preferred, since each page table entry takes memory to track + disk IO is more efficient when a larger amount of data is transferred

Where is page table stored?
- In memory because it can be large
  - page-table base register (PTBR) points to the page table
  - page-table length register (PTLR) indicates the effective size of the page table
  - PTBR and PTLR are stored in the PCB of the process
  - every data/instruction access requires two memory accesses → one for the page table and one for the data/instruction

Translation look-aside buffer (TLB) is a fast-lookup hardware cache in MMU (fully associative memory) that stores page numbers (keys) and frames in which they are stored (values)

- If memory accesses ahve locality, address translation also has locality
- The TLB is small, typically 64 to 1024 entries
- It's typically on the chip, with an access time of a few ns rather than several hundred for main memory

Using TLB

Each process gets it's own page table, but the TLB has entries for multiple processes simultaneously. On a TLB miss, the value is loaded into the TLB for faster access next time. We need to consider replacement policies for this reason.

TLB uses address space identifiers (ASIDs) to determine which entry belongs to which process.

A valid/invalid bit is used for protection. "valid" indicates that the associated page is in the process logical space, and is a legal page. This is in the page table. "invalid" means the page isn't in the process logical address space. We could perform protection with the PTLR.

Context switching?

The OS does the following:

1. saves the PTBR value to the PCB
2. saves the process' page table in its PCB (i.e. copying the TLB to the PCB)
3. marks all TLB entries as invalid (flushes the TLB)
4. restores the PTBR value of the new process
5. restores the TLB as it was saved

Memory access cost with TLB

Let C_{ma} and C_{TLB} be the cost for memory access and TLB respectively.

- Effective access time (EAT) if page table is in memory:

$$\text{EAT} = 2C_{\text{ma}}$$

- Effective access time with a TLB and one-level page table:

$$\text{EAT} = \rho_{\text{hit}}(C_{\text{ma}} + C_{\text{TLB}}) + (1 - \rho_{\text{hit}})(2C_{\text{ma}} + C_{\text{TLB}})$$

- A large TLB improves the hit ration and decreases the average memory cost, for example, let $C_{\text{ma}} = 10\text{ns}$, $C_{\text{TLB}} = 0\text{ns}$. If $\rho_{\text{hit}} = 80\%$ then EAT=12ns (20% slowdown in access time). if $\rho_{\text{hit}} = 99\%$ then EAT=10.1ns (1% slowdown)

Allocation at start of process

If process needs k pages

- if k page frames are free, allocate these frames to pages, otherwise free the frames that aren't needed and allocate them to the process
- OS loads each page in a frame and then puts the frame number in the corresponding entry in the page table
- OS marks all TLB entries as invalid (flushed the TLB) and starts the process. As the process executes, the OS loads TLB entries as each page is accessed, replacing an existing entry if the TLB is full.

Sharing

- Increases level of multiprogramming (code and data shared b/w processes)
- Shared code has to be reentrant, so code using it can't change it

- The OS keeps track of reentrant code and reuses the page frames if a new process requests the same program/library
- Allows running many processes with limited memory
- Paging allows sharing memory across processes, since memory used by a process doesn't need to be contiguous

Comparison with segmentation

- Paging is a big improvement, eliminates the problem of external fragmentation and thus the need for compaction
- Enables processes to run when they are only partially loaded into main memory

Disadvantages:

- Causes internal fragmentation
- translating from virt address to physical is time consuming, the segment table tends to be much smaller than the page table.
- paging requires hardware support (needs TLB)
- paging requires a more complex OS to maintain the page table

Forking in UNIX

- Segmentation and paging allow for more efficient `fork()`
  - copy segment/page table into child's table
  - mark parent and child segments/pages as read-only
  - start child process and return to parent
- Usually child calls `exec()` after fork without attempting to write the read-only segments/pages
  - duplicating the parent's pages isn't usually necessary
- If child/parent tries to write a segment/page, we do this:
  - trap into kernel (?)
  - copy the segment/page (**copy-on-write**)
  - mark both segments/pages as writable and resume execution

Basically, we create a copy of this shared page when it's written to.

Growing the heap/stack

If a process uses memory beyond the end of the stack or allocates more memory than the heap has, a segfault happens, OS allocates more memory, zeros the newly allocated memory (**zero-on-reference**) (done to avoid leaking info), modifies the segment table, and the process is resumed.

Superpages

- The first part of a TLB entry can be:
  - a page number, or a superpage (a set of contiguous pages in one page table) number
- x86 TLB entries can point to 4KB, 2MB, or 1GB pages
- The virtual address can start with a page # or a superpage #
- The SF ends up being smaller and offset much larger

Swap with pages

- Only specific pages are swapped in/out
  - common in linux/windows

Combining segmentation and paging

- Virtual address space is a collection of segments (logical units) of arbitrary sizes

- Physical memory as a sequence of page frames (fixed size)
- Segments are typically larger than page frames
- Map a logical segment onto multiple page frames by **paging the segments**
- In this implementation, a segment must have it's own page table - this supports the sharing of segments, incremental growth of the stack, etc.

Segmented paging addresses

- A virtual address has segment number, page within the segment, and an offset within the page
- Segment number indexes into the segment table, which has the base address of the page table for that segment
- We check the remainder of the address (page no. and offset) against the limit of the segment
- We use the page number ot index into the page table
  - Entry is frame number (like paging)
- We combine the frame number and offset to get the physcial address

Example

Given a memory size of 256 addressable words, a segment page table indexing 8 pages, a page size of 32 words, and 8 logical segments:

1. How many bits is a physical address?
2. How many bits for the seg #, page #, offset?
3. How many bits is a virtual address?

Answers:

1. 8 bits - 256 addressable words
2. 3, 3, and 5 bits - 8 segments, 8 pages, and 32 words page size (offset)
3. 3 + 3 + 5 = 11 bits (virtual address contains seg, page, and offset)

Page Table Structure

- Division of page table into smaller units for very large address spaces e.g. 2^{32} bits
  - Hierarchical paging
  - Hashed page tables
  - Inverted page tables

Hierarchical paging

- break up address space into multiple page tables
- One technique is a two-level page table
  - We page the page table
  - Requires two lookups per memory reference to resolve the physical address

Example:

Consider a 32-bit logical address space with a page size of 4KB (2^{12})

- Since page table is paged, the page number is divided further into a 10-bit page number and a 12-bit page offset.
- Now we have p_1, p_2, and d
  - p_1 is an index into the outer page table (10 bits)
  - p_2 is the displacement within the page of the inner page table (10 bits)
  - d is the page offset (physical) (12 bits)

We can also have more than two levels, e.g. in a 64 bit logical address space, however the number of levels increases to up to 5 (many memory accesses). If we wanted to use three levels, the outer page table would need 32 bits, and wouldn't fit in one page, so we would have to use more levels.

This makes hierarchical paging unsuitable for 64 bit architectures.

Hashed paging

Common in address spaces greater than 32 bits

- The virtual page number is hashed into a page/hash table entry, so page *i* is placed in slow *H(i)* where *H* is an agreed-upon hash function

To handle collisions, each slot is a chain of elements hashing to the same location

- each element contains 1. the virtual page number, 2. the value of the mapped page frame, 3. a pointer to the next element

Virtual page numbers are compared in this chain, searching for a match

- If match found, corresponding physical frame is extracted

Inverted paging

Rather than processes with their own page table to keep track of logical pages, track all physical pages (frames) in one page table

- Inverted page table has one entry for each frame of memory (sorted by physical address)
- Each page table entry consists of a virtual address of the page stored in that real memory location, and an address-space identifier (which process owns that page)

This decreases memory needed to store each page table, but increases the time to search the whole table when a page reference occurs, because the whole inverted page table must be searched (can be large)

- Can use a hash table to limit the search to one or at most, a few page table entries
- TLB can accelerate access as well

Sharing can't be supported with an inverted page table, because one physical page can't have two or more shared virt addresses

Hierarchical vs. Inverted

- Hierarchical page table has one entry per virtual page
- Inverted page table has one entry per page frame (physical frame)

Virtual Memory

Reasoning

- Entire address space of a process is rarely used; process should run even if these aren't in memory
- Only a portion of a process' virtual address space are mapped at any point in time, and the remainder remains in the disk (in files) / in swap space
  - Code pages are stored in a file on disk, e.g. `a.out`
  - data, stack, and heap pages are stored in a special invisible file
  - Some pages are loaded in main memory; so main memory acts as a cache for the disk - page table indicates if a page is on disk or in memory using the valid/invalid bit

Consequences

- Virtual address space isn't constrained by the size of the physical address space
- More processes can fit into memory to run concurrently
- Note that memory accesses must be to pages that are in memory for the vast majority of time, otherwise the effective memory access time approaches the disk access time.

Page fault

- Occurs when page table is accesses and entry doesn't have the valid bit set
- OS must start disk IO to read in the unmapped page
- Page is written back into main memory from disk

When to load page into memory?

- Load all pages of a process at start
  - Wastes memory and IO
  - Virtual address space has to be smaller than physical memory
- Leave to programmer
  - Difficult and error prone
- Request paging - process tells OS before it needs a page and when it is done with it
- Pre-paging - OS predicts which pages will be accessed by process and preloads them
  - Allows more overlap of CPU and IO if OS is correct, otherwise causes page fault
  - Prediction errors can result in removing useful pages
  - Difficult to get right due to code branches
- **Demand paging** - OS loads a page the first time it is referenced (transparent to process)
  - May need to remove a page from memory to make room
  - Process needs to give up CPU while page is loaded (transition to waiting state)

Demand Paging

- Copy of the entire process' memory stored on disk
- Valid bit in page table indicates pages are in memory (v means legal/in memory, i means must initiate disk IO, or invalid memory access)

Page fault?

- Instruction faults on page with invalid
- Page fault causes trap to kernel
  - Saves registers and state of faulting process
  - Checks if memory reference was legal and determines location of page on disk
- OS puts faulting process on wait queue of disk, and starts reading unmapped page into a frame
  - Selects a free page frame (or a page to replace with a replacement algo) and zeros it (**zero-fill-on-demand**)
- When IO being done, OS context switches to another process
- OS gets an interrupt when done
  - Saves regs and state of other process
  - Updates page table entry to indicate the page table is now in memory
- OS puts faulted process back into ready queue
  - Resumes interrupted instruction when the CPU gets allocated to this process again

Swap space

- When page removed, if it contained code, it can just be deleted, but if it had data, it needs to be saved into swap space
  - Swap IO is faster than file IO because swap is allocated in larger chunks and requires less management than a filesystem
- A page can exist in physical memory, the filesystem, or in swap space
- Additional bits are needed in the page table for where the page can be found

Demand paging works because in practice, processes typically exhibit locality of reference:
- **temporal locality** - if a process accesses an item in memory, it will tend to reference the same item again soon
- **spatial locality** - if a process accesses an item in memory, it will tend to reference an adjacent item soon

Performance

Let ρ_{fault} be the probability of a page fault ($0 \leq \rho_{\text{fault}} \leq 1$)
- EAT is

$$(1 - \rho_{\text{fault}})C_{\text{ma}} + \rho_{\text{fault}}C_{\text{pagefault}}$$

  where C_{ma} is the memory access time and $C_{\text{pagefault}}$ is the page fault service time, equal to servicing the interrupt, reading the page, and restarting the process.
  - Disk IO is expensive (4 orders of magnitude more than memory access)

If memory access time is 200ns and average apge fault service time is 25 ms,

$$\text{EAT} = (1 - \rho_{\text{fault}}) * 200\text{ns} + \rho_{\text{fault}} * 25,000,000\text{ns} = 200\text{ns} + \rho_{\text{fault}} * 24,999,800\text{ns}$$

If we want EAT to be only 10% slower than memory access time, what ρ_{fault} do we need?
- $(1 - \rho_{\text{fault}}) * 200 + \rho_{\text{fault}} * 25,000,000 = 220 \rightarrow \rho_{\text{fault}} \sim 10^{-6}$
- If page fault ratio gets larger, effective access time approaches the disk acces time!!

How does OS safely restart a faulting instruction?
- Need hardware support to save the faulting instruction and CPU state
- Instructions with side effects?
  - Page fault can happen in the middle of an instruction
  - e.g. `mov a, (r10)+` moves `a` into the address contained in `r10` and increments `r10`
  - Solution is to undo all side effects
    - delay side effects until after all memory references are valid

Page replacement

- Swap daemon handles swapping when process faults and memory is almost full
- Handling a page fault now needs 2 disk accesses

Algorithms

FIFO

- Throw out oldest page
  - Can easily happen to pages used frequently
- Visual representation of page faults and CPU cycle (I think this will be on exam)
  - Table with referenced page as column identifier, frame number as row identifier.
  - Depending on algo, we add referenced page frames in, starting at 0.
  - Once full, algo decides what gets replaced, we can mark page faults when a referenced page isn't in the page table and (e.g. for FIFO), the oldest page is thrown out
- Adding memory may not help with FIFO, it depends on the reference stream, sometimes we can get more page faults with more memory for a different stream of page references. Called **Balady's anomaly** - increase appears from $3 \rightarrow 4$ frames.

Optimal (OPT)

- Look into future and throw out page accessed furthest in the future
- Provably optimal but hypothetical
  - gives an upper bound on hit rate; (no. hits) / (no. hits + no. misses)

Least recently used (LRU)

- Throw out page that has not been accessed in the longest time
  - Use the recency of it's access
- An approximation of OPT
  - Works well if the recent past is a good predictor of the future (locality)

LRU implementation

- Needs hardware support
  - Approach 1 (timestamp): keep a timestamp for each page with time of last access and kick out the LRU page
    - too expensive! OS needs to log timestamp for each memory access, and compare timestamps of all to evict a page
  - Approach 2 (stack): keep a doubly linked list of pages, where front is most recently used, and end is LRU; move corresponding page to front of list on memory access
    - still too expensive, OS must modify 6 pointers on each memory access (worst case)
- Approximating LRU
  - Use a reference bit or bits on each page - hardware sets bit on each access
  - Set to 0 at some time
  - For multiple bits, right shift the vector at some interval to decrease it over time
  - lowest numbered page is kicked out on page fault
  - doesn't guarantee order of the pages
  - page fault still needs search through all the pages

Second-chance page replacement

- Use one reference bit per page
  - on page fault, OS checks reference bit of page the pointer refers to

- if reference bit is 0, replaces page and advances to next page to be considered for replacement (OS keeps a circular list of pages, and pointer to next page to consider)
- if reference bit is 1, clears reference bit and advances to check next page

Disadvantage - less accurate than additional-reference-bits algo since reference bit only indicates if page was used at all since last time it was checked.

Advantage - Fast since it requires setting a single bit on each memory access (no need to shift) and handling a page fault is faster as we only search the pages until we find one with a ref bit that isn't set.

This algorithm is called the clock algorithm - partitions pages into two categories, young and old. Only two categories is faster than comparing *how* young a page is. There is a case where it's cheaper to replace a page that is young - if it hasn't been modified; no need to write it out.

Enhanced second-chance

- Keep a modified bit and a reference bit
  - '1' means page is modified
  - '0' means page is the same as on disk
- (r, m) has 4 possibilities
  - (0, 0) - not used or recently modified, best for replacement
  - (0, 1) - not used, but recently modified, not so good because OS must write it out
  - (1, 0) - recently used, not modified, probably used again, but OS doesn't need to write before replacing it
  - (1, 1) - recently used and modified, not good candidate

OS searches for (0, 0), in at most three passes:

1. - replace (0, 0) if exists
   - If (0, 1), initiate IO to write out page, clear modified bit, and continue
   - If ref bit set, clear it
   - If no page replaced, no (0, 0) existed
2. - A page that was (0, 1) or (1, 0) may be (0, 0) now; replace it if so
   - If the page is being written out, wait for IO, then remove page
   - A (0, 1) page is treated same as first pass
3. - All pages will be (0, 0) by this pass

Counting based algos

- Count number of references
  - least frequently used (LFU) throws out page with smallest ref count
  - most frequently used (MFU) throws out page with highest ref count
- These are expensive and don't approximate OPT or LRU

Frame Allocation

- processes need a min number of frames to hold all pages a single instruction can reference
  - otherwise, a page fault occurs in the middle of instruction execution
- allocation algos
  - **equal allocation** - give each process an equal share of available frames in system
    - e.g. 128 frames in memory, kernel needs 35, free frame pool has 3 frames, and 5 active processes in the system, each gets 18
  - **proportional allocation** - allocate available frames to process according to size or priority

$$s_i = \text{size of process } p_i$$

$$S = \sum s_i$$

$$m = \text{total number of frames}$$

$$a_i = \text{allocation for } p_i = \frac{s_i}{S} * m$$

Allocation policies

- **Global replacement** - each process selects a replacement frame from the set of all frames in the system
  - one process can take a frame from another
  - a process can't control its own page fault rate, which depends on the other processes
  - execution time varies
  - thrashing becomes likely
- **Local replacement** - each process selects from only its own set of allocated frames
  - more consistent per-process performance, but memory can be underutilized

Thrashing

- Process is thrashing when it spends more time paging than executing
- Happens when pages are continuously kicked out (global page replacement) when they are still in use
- We can avoid this by using per-process replacement (local); each process has it's own pool of pages large enough to avoid thrashing
- working-set model approximates the program locality - set of pages that are actively used together
- working set is the set of all pages a process referenced in the past Δ time units
- If Δ is too large, it can overlap other localities and can result in thrashing
- If Δ is too small, it might not encompass entire locality
- Working sets are expensive to compute

Alternative is to track page-fault frequency of each process and use it as feedback (ctrl signal)

- give process more page frames if fault frequency > some threshold
- take away page frames if fault frequency < some other threshold
- More consistent performance independent of system load

We want the system-wide mean time between page faults is equal to the time it takes to handle a page fault

File System

Disks

- Array of blocks, where each block is a fixed size array
- A disk with a filesystem is a *volume*

File abstraction

- Named collection of bytes/blocks recorded on secondary storage
- Can be structured or unstructured (stream of bytes, or XML, JSON)
- File structures (e.g. executable vs text)

Two kernel data structures

System-wide open-file table - doubly linked list of files in use; shared by all processes with an open file

- contains an open count (how many processes have it open)
- file attributes
- file location on disk
- pointers to location of file in memory

Per-process file descriptor table - doubly linked list of pointers into the open-file table for files opened by the process; stored in PCB

- pointer to its entry in system-wide open-file table
- current-file-position pointer (offset into the file)
- mode the process accesses the file (r, w, rw), and file locks (shared and exclusive)
- accounting info

File open

- File name and access mode as input
- Search directory for named file
- Check access mode with file's permissions
- If permitted, copy the entry to the **system-wide open-file table** if not already opened by another process, and increment the open count
- Add a pointer to open-file table entry in the process' file descriptor table
- Initialize the curent-file-position pointer to the start of the file
- Return a pointer to the file entry in the process **file descriptor table** (file descriptor)
  - When a file operation is requested, the file is specified via an index into the FD table, which tiself links to the corresponding entry in the system-wide open-file table
  - Eliminates need to search the directory each time (one time cost)

File close

- Locate and remove file's entry from process file table
- Decrement the open count in the system-wide open-file table
- If open count becomes zero, remove the entry from system-wide open table
  - Otherwise, kernel might run out of space

Operations with a file descriptor

- `fd = Open("filename")`
- `Close(fd)`
- `Truncate(fd)`
- `Seek(fd, offset)`
- `Read(fd, buffer, length)`

- `Write(fd, buffer, length)`

File access methods

- Sequential access - info in file is processed in order
  - based on tape model - pointer to the next byte in the file
  - update the pointer on each read/write/seek operation
  - commonly used in text editors, compilers
- Direct access - address any block in the file rapidly given its offset within the file
  - Is based on the disk model - disks allow random access to any block
  - Block num to access is relative to beginning of file
  - Possible to read block 8 then 12 then 4
  - Used by databases - compute the block with the answer (e.g. hashing key) and read the block directly

It's easy to implement sequential access on a direct access system, but using sequential access for a direct access system is extremely inefficient

On-disk filesystem data structures

- **boot-control block** contains info needed to boot OS from the corresponding volume and is typically the first block of a volume
  - might not exist if no OS present
- **volume control block** contains info about the volume, such as block count, block size, free-block count, free-block pointers, free-FCB count, FCB pointers, etc.
  - Called a superblock in UFS and master file table in NTFS
- Per volume **directory structure** contains mapping between file names and inode numbers
- Per-file **file-control block** (FCB) contains process-independent info about the file, including a unique identifier allowing association with a directory entry, ownership, permissions, file size, pointers to file data blocks, etc.
  - Called an index node or inode in the UNIX file system (UFS)

In-memory filesystem data structures

- **mount table** contains pointer to volume control block of each mounted file system
- **cached directory structure** holds directory information for recently accessed directories
- **system-wide open-file table** contains a copy of the FCB of each open file and other information like the open count
- **file descriptor table**, a per process open-file table that contains a pointer to the entry in the system-wide open-file table
- **buffers** holding blocks that are being read or written to disk
  - a process writing to disk actually writes to a buffer, the OS writes buffered data to disk asynchronously when convenient
  - a process reading to a disk is actually reading from a buffer; OS may read blocks from disk ahead of time to fill the buffer!!

Directory

Symbol table used to translate file names to an index node number, to get a pointer to FCB (**file-control block**), lookup the file name (unless the fd is availble). The directory creates the namespace of files, each volume contains information about files in the *volume table of contents* or *directory structure*

- A directory structure organizes all files and directories into a large tree.

Implementations

Implemented as a list of directory entries (**dentry** nodes) where each entry is a file anme and its associated index node number

- Simple to program but slow if implemented with a linked list (linear search time)
- could keep the list sorted and use binary search to finda file name (complicates deletion and creation of files)
- could use a balanced tree for best performance

Hash table: linear list of directory entries with a hash table used to get a pointer to the corresponding directory entry in the list.

- Decreases dir search time
- collisions need to be handled
- each location can be a linked list of entries

Operations

- Search for file
- Create file
- Delete file
- List a directory
- Rename a file
- Traverse file system

Single-level dir structure

One namespace for entire disk, filenames must be unique, hard with many files, and can make file names confusing. Directory structure is held in a special area of disk, and files can't be grouped into subdirectories.

Two-level dir structure

Each user has their own directory, the UFD (user file directory). Can have the same filename for different users, but a user's files are unique. MFD (master file directory) is indexed by user name or uid and each entry points to UFD of a user. To create or delete, UFD of the user is searched. Files can't be grouped into subdirectories. Pathname is defined by the username and filename (unique).

Multilevel dir structure

Allows users to create their own subdirectories, each directory is a special file that can contain other files - one bit used to indicate directory or file (0). Namespace is tree structured, used in UNIX. Directories stored on disk with flag bit, user programs can read directories but special calls can write dirs. Each process has a current directory, and there is one special root directory. Each directory contains dentry nodes, in no order.

- Absolute pathname begins at root and files a path to specified file

Symbolic links

- A dir entry can be marked as a link, used for sharing
- Soft or symbolic links (`ln -s`) can be used, makes a symbolic pointer from one file to another

$$B \to A$$

  - Removing B doesn't affect A
  - Removing A leaves the name B in the dir, but contents don't exist (dangling)
- When FS encounters a symlink, automatically translates

- **problem** with circular links that create infinite loops, e.g. listing all files in a directory and its subdirectories. (solve by limiting number of links traversed)
- Hard links add a second connection to a file
- OS maintains reference counts for files in FCBs, so it will only delete a file after the last link to it has been deleted
- **problem** with user creating circular links with dirs, then OS can never delete the disk space
  - solution is to allow no hard links to directories

Protection

- OS has to allow users to control sharing of files, and grant or deny access to file operations depending on access control info
  - Each file/dir gets an access control list (ACL), specifying user names and types of access allowed for each users
  - problems:
    - constructing ACL is tedious with many users
    - need to know all users in advance
    - size of direntry no longer fixed, complicates space management
- Condense length by using user classess
- In UNIX, three categories of users (owner, group, public)
  - A field of 3 bits for 3 access privileges
  - Maintain a bit for each privilege
- We have three sets of bits for r/w/e for o/g/p (chmod 777 = 111111111)

Disk Structure

Disks are organized in disk packs with a stack of circular platters, each has two sides, each side has a read/write head. Tracks are concentric rings on a disk platter, and bits are laid out serially on tracks.

A cylinder is a set of vertically aligned tracks on all platters, each track is divided into sectors (sector/block is minimum unit of data transfer.

Free space management

FS neesd to allocate space released after deleting files to new files - must be able to find free space quickly. To keep track of free blocks, FS maintains a free-space list, when a block is allocated to a file, they are removed from a list, when deleted, freed blocks added to list.

Free space list might not be a list. We also need to maintain a free-inode list.

Bit vector

Each disk block is represented by a bit in the free space list vector. To find a free block, the FS compares each word in the bitmap with zero. First non-zero word is scanned for the first bit 1 after. It's expensive to find a free block if most blocks are used.

Bitmap can be too big to keep in memory, e.g. block size = 512 bytes, disk size = 2GB, n = $\frac{2^{31}}{2^9} = 2^{22}$ blocks, or 2^{22} bits (512KB)

Linked free-space list

- Link together all free disk blocks, each contains a pointer to the next free block. The head of the list gets cached in memory.
- No need to traverse the entire list, if the no. of free blocks is recorded.

- This way, we only need to keep track of the head of the free-space list (and no bit vector is needed).

problems:
- linked list can get disorganized over time, and we can't get contiguous space easily

Extensions
- Grouping
  - Modify linked list to store address of next n-1 free blocks in the free block, plus a pointer to next block that contains free-block-pointers (like this block)
- Counting
  - Space is contiguously used and freed frequently, so
    - keep the address of the first free block and the number of following free blocks
    - free-space list has entries containing addresses and counts

Reliability

Single FS op might need updating multiple physical disk blocks (inode, data blocks, bitmap, etc.)
- To move a file b/w directories, it has to be deleted from the old directory and added to the new directory.
- To create a new file, file system must allocate space on disk for file's data, write the new inode to disk, add the file number and name to the directory that contains the file.

Problems
1. Physical disk blocks are updated one at a time
   - crash may occur between multiple updates causing inconsistencies among filesystem data structures
2. **write-through** is really slow (writing modified data back to disk in a synchronous fasion), so updates are cached in memory and written to disk when convenient
   - crash might occur before disk IO, causing a loss of data

Consistency

UNIX uses write-back strategy (delay writing modified data back), with periodic forced writes. Other processes read data from cache rather than disk, there's a potential for loss of 30s worth of cached changes (if writing back every 30s). The user can use `sync` to flush the cache to disk immediately.

How to write changes made to a file?
- Naive approach:
  - delete old version; create new version
- Correct approach:
  - Write new version in a temp file, move old version into another temp file, move the new version to the real file, unlink the old version.
  - On a crash, look at the temp area, and if there is any files out there, tell the user there might be a problem

Metadata updates - recover after crash by checking if there were any in-progress operations, use a status bit to indicate metadata is changing, clear the bit when it's done. If it's still set, use a consistency checker (`fsck` in UNIX), which scans the entire disk for inconsistencies and fixes them. Used in FAT and UNIX file system.

Issues with the allow break and repair approach is that it's time consuming and might not be successful. Also difficult to reduce every operation to a safely interruptible sequence of writes, and to achieve consistency when multiple operations occur simultaneously.

Example

Extending a file by one block; operations required:

1. find free block
2. set the bit in free-space bitmap
3. update inode with pointer to free block and new file size
4. write data to the new block

When a crash happens:

- If bit set in the bitmap but a pointer to the block isn't added to any inode, writing the inode must have been in progress when the system crashed.
- If a bit is set in the bitmap and a pointer to this block is added to an inode but file data isn't written to that block, writing data must've been in progress

Transaction concept

A transaction is a group of operations that are atomic and durable

- atomic means performed as a group or not at all
- durable means future failures don't corrupt previously committed transactions

We make a set of metadata updates tentatively, if we don't get to commit, due to a crash, then roll back the updates as if it never happened.

- Commit makes the transaction durable by writing a single sector on disk (we assume this happens atomically)

Idea is borrowed from database systems and adopted in **log-based transaction-oriented systems**

Write-ahead

- Write operations performed sequentially and synchronously in an on-disk data structure (log, journal, intention list)
  - Ignore changes if crash occurs in the middle of these operations
  - note that sequential IO is faster than random IO, can be done synchronously
- Commit the transaction when all changes are on log
- Write the changes asynchronously to appropriate blocks of disk
  - Use ptr to indicate what changes in log are written successfully
  - If crash occurs after commit, replay the log starting from that pointer to make sure the updates make it to the disk
- Journaling eliminates the need for doing post-crash file system consistency check (`fsck`)
  - This is a general solution to the reliability problem, but
  - Data is written twice

UNIX filesystem

- Boot block (boot program which loads OS)
- Superblock which defines the FS
  - Size of FS and of inode list
  - list of free disk blocks (and index of next free block)
  - list of free inodes (and index of next free inode)
  - location of inode of root directory (inode 2)
- inodes containing file metadata; each identified by a uint.
  - can translate these to a location on disk
  - inodes are either put together as a group (in inode list), or spread across the disk
- File data blocks

names/dir structure

- Directory entry is a collection of (name, inode no.) pairs for files and directories within that directory
  - Is stored as a regular file
  - Only OS can modify it
  - . and .. are stored as ordinary file names with inode numbers pointing to the inodes of the same directory and the parent directory

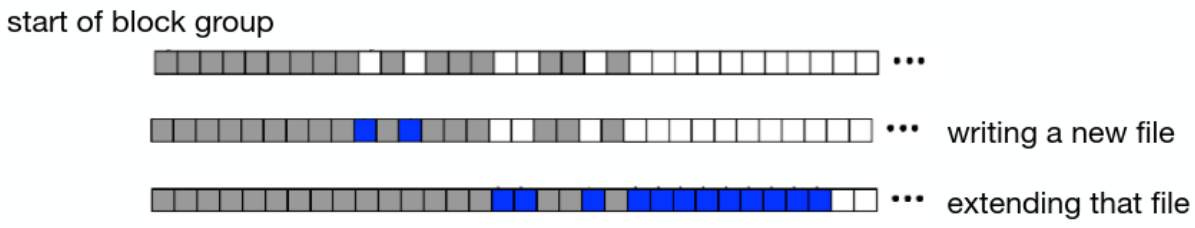
FFS

Data locality

- A block group si a set of contiguous cylinders (on disk) - we can seek fast between cylinders in the same block group
- Block group allocation - files in the same dir are in the same group, subdirectories are in different block groups
- Inode list and bitmap are spread throughout the disk and placed near file blocks
- 10% of disk space is reserved so we can allocate data blocks in correct cylinder groups (free space scattered across cylinders)

FFS first-fit block allocation

- When allocating space to a new file, search for a free block from the start of the group
- When extending a file, search for a free block from the last block allocated to that file; if not enough exist, allocate the remaining blocks from a new range
- There will be a few little holes at the start and a big hole at the end of a block group, so small files will be fragmented, and large files will be mostly contiguous



NTFS

- Master file table stores metadata and data
- Directories are organized with B-trees
- Index structure is a variaible-depth tree

- Variable length extents (adjacent disk blocks)
- Journalling for consistency checking

Secondary storage

- NVMs and HDDs
- Disk looks like a linear array of blocks that can be read from or written to
- For disks, accessing two nearby blocks is usually faster than two that are far apart, due to less head movements
- Accessing disk blocks sequentially is typically faster than a random pattern (less head movements)

Disk performance

Rotation

- Delay is the time necessary for the spindle to rotate the desired sector to the disk head, depends on speed of disk spin
- e.g. disk head currently positioned over sector 6
  - average case: if rotational delay is R, disk has to incur a rotational delay of R/2 to wait for sector 0 to come under the read/write head
  - worst case: wait for sector 5 to come under the read/write head (almost a full rotation)

Seek

- Seek time is the time to move the disk arm to the desired cylinder/track (b/w 3ms and 12ms)
- e.g. move the arm out to a cylinder or in to a cylinder

Disk I/O

- Positioning time is seek time + rotational latency
- Transfer rate (bandwidth) is data flow rate from disk to computer
  - typically 100s of Mb per second
- Transfer time = amt of data / transfer rate
  - e.g. rate is 100Mb/s, 5ms to transfer 512Kb of data
- Avg. disk IO time = avg. positioning time + transfer time (+ ctrl overhead)
- Disk I/O rate is the amt of data to be transferred divided by the disk I/O time

Example:

- Average I/O time to transfer a 7LB block on a 7200 RPM disk with 5ms average seek time, a 0.1ms controller overhead, and 1Gb/s transfer rate

$$\text{avg. rotational delay} = \frac{1}{2 * \frac{7200 \text{ rotations}}{1 \text{ min}} * \frac{1 \text{ min}}{60000 \text{ ms}}} = 4.17 \text{ ms}$$

$$\text{avg. seek delay} = 5 \text{ ms}$$

$$\text{controller overhead} = 0.1 \text{ ms}$$

$$\text{transfer time} = \frac{4\text{KB}}{\frac{1 \text{ Gb}}{\text{s}} * \frac{1 \text{ GB}}{8 \text{ Gb}} * \frac{1024*1024 \text{ KB}}{1 \text{ GB}} * \frac{1 \text{ s}}{1000 \text{ ms}}} = 0.03 \text{ ms}$$

$$\text{avg. I/O time} = 4.17 \text{ ms} + 5 \text{ ms} + 0.1 \text{ ms} + 0.03 \text{ ms} = 9.30 \text{ ms}$$

- HDDs can also have a cache

- Write-back - request to write data can be acknowledged right away by putting the data in cache and writing to sectors later
- Write-through - request to write data is acknowledged only after the data is actually written to disk

Reducing access time

- Sequential I/O is optimal on HDD
  - disk seek is only necessary at the end of a track and need to move to beginning of another
  - random access causes disk head movement and is slower

1. make disks smaller and spin faster
2. decide the order in which IO requests are issued to minimize head movement
3. layout data on disk so that related data are on nearby tracks
   - e.g. file contents near its metadata
4. choose sector size carefully
   - smaller sectors means more disk seeks for the same amount of data

Non-volatile memory (NVM)

- These are electrical rather than mechanical, like an SSD
  - No disk head, so no need for complex scheduling (usually FCFS or variants), unlike HDD
  - random-access I/O is a lot faster compared to disk
  - writing to NVM is slower than reading (blocks need to be erased before write)
- SSDs are built with transistors
  - like USB drives
  - more reliable than HDDs because no moving parts
  - faster than HDDs because no seek time or rotational latency
  - more energy efficient than HDDs
  - more expensive

NAND semiconductors

- Organized into banks
  - blocks in each bank, pages in each block (like sector in HDD)
- Data can't be overwritten, so to write to a page, you have to erase the entire block first
- lifetime is approx. 100000 program erasse cycles
  - measured in DWPD (drive writes per day)
  - 1TB NAND with 5 DWPD is expected to have 5 TB per day written in warranty period
  - frequently erased blocks wear faster, so controller does wear levelling by erasing and overwriting less erased pages

Disk Scheduling

HDD

- FCFS orders the queue of IO by time arrived
  - fair but not necessarily fastest
  - best when light disk load
- SSTF is shortest seek time first; order queue by track, pick requests on the nearest track to service first
  - always go to next closest track
  - not optimal - minimizes head movement in each step but not total head movement

- Issues:
  - drive geometry isn't available to the OS (could do NBF instead - nearest block first)
  - Starvation is possible, e.g. a stream of requests to inner track, but outer track is waiting
- SCAN scheduling
  - Disk head continuously scans back and forth across the disk
  - Each pass is a sweep
  - We need to know the head's initial position and the direction of its movement
  - If there is no request between the current position and the disk end, we can change the direction immediately (called LOOK scheduling)
- C-SCAN scheduling (circular)
  - After reaching one end, immediately travel to the start without without servicing requests on the way
  - C-LOOK goes from request to next request rather than disk end to start, once end is met.
  - Treats disk cylinders as a circular list, wrapping around from final cylinder to the first one
- SCAN and C-SCAN perform better when there's a heavy load on the disk
  - starvation is less likely compared to SSTF with these scheduling algorithm
- Compared to C-SCAN, SCAN favors middle cylinders (passes thru middle twice before coming back to the outer cylinder
- All these algorithms ignore rotation delay and optimize for seek delay only
  - Makes sense if seek much higher than rotational delay, but maybe not the case for some tech

Considering both seek time and rotational latency

- To approximate SJF policy, it's necessary to condier both seek time and rotational latency
- Shortest time first (SATF)
  - service the I/O request with the least positioning time (seek + rotational latency)
  - need to know relative time of seeking compared to rotation (OS doesn't usually have this info)
- Like traveling salesman problem (NP) - hard
- In modern systems, disks have sophisticated internal shcedulers themselves
  - so the OS scheduler usually issues a small batch of I/O requests that it thinks are the best
  - internal disk scheduler services these requests in the SATF order

Read-ahead

- To reduce the number of seeks, we can read blocks that will probably be used while they're under the head
- Read blocks from disk before user's request and place buffer on disk controller

Deadline scheduler

- Maintain separate read/write queues (specifically two read and two write) and give read reqs. a higher priority
  - processes are more liekly to block on read than write as **write-back** is often used
  - requests are submitted to queues in batch
- One read queue and one write queue are kept sorted by logical block address order
  - C-SCAN as both queus are sorted in logical block addr. order
- One read queue and one write queue are kept sorted by FCFS
- For each batch, if there are requests in FCFS queues older than some predefined age, they are serviced first (fairness and timeliness); otherwise, requests in queues sorted by logical block address are serviced.

RAID

- Partitioning is dividing a disk into block groups
  - Each partition is treated as if it were a separate drive, so ti can hold a file system or beused as a raw disk (e.g. for swap space)
    - Partition info written at a fixed disk location
    - A device entry is created for each disk partition
  - A device with a boot partition is called a boot disk

Formatting

- Low-level formatting is marking out tracks and creating sectors on the disk (manufacturing)
- Logical formatting is writing initial data structures of a filesystem (freespace list, free inode list, etc.) on a volume
  - Mounting is the process of making the file system in a partition (i.e. a volume) available to system and users

Swap space management

- Swap space can be carved out of filesystem or be in a separate partition
- If in a raw partition, a separate swap storage manager allocates and deallocates disk blocks, optimizing for I/O speed
  - advantage: better performance than swapping in filesystem because FS services are bypassed
  - disadvantage: adding more swap space requires repartitioning the disk

Error detection and correction

- **ECC** (error correction code) is a form of redundancy introduced to detect and correct some errors
- calculated for every disk sector at write time and stored on disk; then recalculated at read time to detect and correct potential problems
  - if only some are corrupted, a soft error si signalled and the error will be corrected
  - if more bits are corrupted, a non-correctable hard error is signalled
- an error can indicate a bad sector
  - disk controller stores the list of bad sectors and performs sector sparing (replacing bad sectors with spare sectors from a reserved pool)

RAID 0

- disks in parallel to increase file IO bandwidth

More disks = more failures:

- if p is probability of disk failure, the probability of at least one failure given N disks is $1 - (1-p)^N$
- e.g. if p=0.01, then with 100 disks, probability of at least one failure is 0.634
- We can partition and redundantly spread data across drives to improve reliability (RAID 1 to RAID 5+)

RAID = Redundant Array of Independent Disks

- Can speed up IO
- More space for persistent storage
- RAID makes data less vulnerable to losses, with redundancy
- Looks like a big fast reliable disk to the file system
  - when a FS issues a IO request, the RAID internally must calculate what disks to access to service the request, and subsequently issue physical IO commands

- At a high level, it's like a computer system with a microcontroller, memory, and disks (**storage as a system**)

Fail-stop fault model
- A disk can be working or failed at a time; a failed disk can be immediately detected
  - a working disk: all blocks can be read or written
  - a failed disk - permanently lost data
- Disk corruption and bad sectors aren't defined with this model

Evaluating design
- capacity: how much useful capacity is available to users of a RAID system comprised of N disks, each having B blocks: $N * B$ without redundancy, $\frac{N*B}{2}$ with mirroring
- parity based schemes are somewhere between this

RAID level 0
- Striping
- Spread logical data across mutliple disks; one by one spread of blocks across disks
  - no redundancy, just parallel disks
  - high data transfer rate, low reliability
  - Disk = A % no_disks, Offset = A / no_disks
- Capacity is $N * B$, low reliability, disk failure = data loss
- Performance is good because disks can be used in parallel to serve IO requests
  - bandwidth is N times the bandwidth of a single disk as all disks are used in parallel
  - latency of a single-block request should be identical to that of a single disk because request is redirected to one of the disks that contain this block

Chunk sizes
- A small chunk size means many files are striped across many disks
  - increasing parallelism of reads and writes to a single file

RAID level 1
- Store more than one copy of each block on separate disks to tolerate disk failures
  - usually two physical copies of each block (duplicates each block)
  - to read, RAID can read either copy - so it can choose the closest copy
  - to write, RAID updates both copies of data - can happen in parallel
  - high reliability
  - high storage overhead, low write bandwidth, expensive
- Capacity is $\frac{N*B}{2}$ because only half of total disk capacity is used
- Better reliability, can tolerate failure of up to N/2 disks if they don't contain the same data
- Performance:
  - Read latency is the same as latency on a single disk as RAID only forwards the request to a disk that contains one of the two copies
  - write latency is the maximum of the write latency of the two disks because the two writes must happen in parallel
  - steady-state write throughput: max bandwidth obtained during sequential (and random) writing to a mirrored array is half of the peak write bandwidth of RAID level 0
  - steady-state read throughput: the max bandwidth obtained during random reading from mirrored array is full bandwidth, but max during sequential read is lower than peak read bandwidth

RAID levels 2 and 3

- Add parity bits to data striping
- Bit-interleaved parity (2 and 3) interleave at the level of bits, i.e. store the Hamming code (RAID level 2) or the parity bit (RAID level 3) in the parity disk
- Block-interleaved parity (RAID level 4) - interleave at the level of blocks, i.e. store the parity block in the parity disk

RAID level 4

- Data striped across multiple disks
  - successive blocks are stored on successive (non-parity) disks
  - increased bandwidth over a single disk
- parity block is constructed by XORing data blocks in a stripe
  - $P0 = D0 \oplus D1 \oplus D2 \oplus D3$
- Computing parity bit causes writes to be slower
  - done by the raid controller in its cache (offloads parity computation from CPU to controller)
- Recovering data stored on a disk that failed
  - if one sector of D2 is damaged it can be replaced with $D2 = D0 \oplus D1 \oplus P0 \oplus D3$

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

- For reads smaller than a block size, need to access only one disk (better than raid 3)
- For small writes
  - read the current stripe of blocks, compute parity with the new block, write the parity block
  - better solution is to read the current version of block, read current version of parity block, compute how parity would change, if a bit on block changed, then the corresponding parity bit needs to be flipped, write the new version of block, write the new version of parity block
- For large writes, compute the parity block and store on the parity disk
- Disk containing parity block is updated on all writes (loaded heavily)

RAID level 5

- Data and parity blocks distributed across disks
  - spreads load evenly
  - multiple writes can potentially be serviced at the same time
  - all disks can be used for serving read requests
- Any one disk can fail and data reconstruction is still possible
- RAID 5 vs. normal disk
  - RAID 5 is better throughput, better reliability, bandwidth for large reads, small waste of space
  - normal disks perform better on small writes
- RAID 1 vs. RAID 5

- RAID 1 wastes more space (storage overhead)
- for small writes: RAID 1 is better

more

- RAID 6 allows 2 disks to fail by using 2 blocks of redundant data
- must do something more complex than XOR, we use error corecting codes (EVENODD code e.g.)

Virtualization

- Simulating interface of a physical object, allowing multiplexing or aggregation
- OS virtualizes CPU, memory, etc.

VMM (virtual machine monitor)

- Also called hypervisor, sits between OSes and hardware, giving each OS the illusion that it's controlling or directly interacting with hardware
  - in reality, VMM is in control of hardware and multiplexes the running OSes across physical resources
  - transparently virtualizes the hardware underneath the OSes
- A VM is an OS and its applications that sit on top of a VM
  - VMM is a simple OS for VMs

Terminology

- host is the underlying hardware
- Guest is the OS provided with the virtual copy of the host

Benefits and features

- Suspending, transferring, resuming a VM
- Templating (creating multiple instances of the same VM)
- Live migration - balance load and improve performance