Files

Each file has a type indicating its role in the system. ***Regular files*** contain arbitrary data, ***directory*** is an index for a related group of files, and a ***socket*** is for communicating with a process on another machine. Other file types (beyond our scope): ***Named pipes*** (FIFOs), ***symbolic links***, and ***character and block devices***. Note that applications often distinguish between ***text files*** (regular files with only ASCII or Unicode characters) and ***binary files*** (everything else).

All files are organized as a hierarchy anchored by ***root directory*** named / (slash). The kernel maintains ***current working directory (CWD)*** for each process and is modified using the ***cd*** command. Locations of files in the hierarchy denoted by ***pathnames***. ***Absolute pathname*** starts with '/' and denotes path from root (i.e., ***/home/droh/hello.c***). ***Relative pathname*** denotes path from current working directory (i.e., ***../droh/hello.c***). We use ***lseek()*** to change the current file position (seek) which indicates the next offset into file to read or write.

Opening a file informs the kernel that you are getting ready to access that file. It returns a small identifying integer ***file descriptor*** (fd == -1 indicates that an error occurred). Each process created by a Linux shell begins life with three open files associated with a terminal:

1. 0: standard input (***stdin***)
2. 1: standard output (***stdout***)
3. 2: standard error (***stderr***)

Closing a file informs the kernel that you are finished accessing that file. Closing an already closed file is a recipe for disaster in threaded programs. Always

check return codes, even for seemingly benign functions such as close().

Reading a file copies bytes from the current file position to memory, and then updates file position. It returns number of bytes read from file fd into buf. The return type ssize_t is a signed integer (nbytes < 0 indicates that an error occurred). ***Short counts*** (nbytes < sizeof(buf)) are possible and are not errors! Short counts can occur when encountering (end-of-file) EOF on reads, reading text lines from a terminal, or reading and writing network sockets. The best practice is to always allow for short counts. ***Writing*** a file copies bytes from memory to the current file position, and then updates current file position. It returns number of bytes written from buf to file fd (nbytes < 0 indicates that an error occurred). As with reads, short counts are possible and are not errors!

The UNIX kernel represents open files using two descriptors referencing two distinct open files. Descriptor 1 (***stdout***) points to terminal, and descriptor 4 points to open disk file. There is one ***descriptor table*** per process, an ***open file table*** that is shared by all processes that the descriptor table points to, and a ***v-node table*** that is shared by all processes that the open file table points to.

File sharing occurs when two distinct descriptors sharing the same disk file through two distinct open file table entries (e.g., calling open twice with the same filename argument). A process can also share files using the fork function. A child process inherits its parent's open files, and after the fork the child's

table is the same as the parent's, and +1 to each reference count.

A shell implements I/O redirection by using ***dup2(oldfd, newfd)***, which copies (per-process) descriptor table entry oldfd to entry newfd.

Virtual Memory (VM)

A page hit is a reference to a virtual memory word that is in physical memory (DRAM cache hit). A page fault is a reference to a virtual memory word that is NOT in physical memory (DRAM cache miss). To handle a page fault, the page fault handler selects a victim to be evicted and the offending instruction is restarted, leading to a page hit.

Sharing Code/Data Process:

1. Process 1 maps the shared object
2. Process 2 maps the shared object
    a. ***NOTE:*** The virtual addresses CAN be different
3. Two processes mapping a private ***copy-on-write (COW)*** object
4. The area is flagged as private copy-on-write
5. Page table entries in private areas are flagged as read-only
6. Instruction writing to private page triggers a protection fault
7. Handler creates new R/W page
8. Instruction restarts upon handler return
9. Copying deferred as long as possible

VM areas initialized by associating them with disk objects, which is known as ***memory mapping***. The area can be backed by a ***regular file*** on disk (e.g., an executable object file) whose initial page bytes come

from a section of a file. Or it can be backed by an ***anonymous file*** (e.g., nothing), where the first fault will allocate a physical page full of 0's (demand-zero page), and once the page is written to (dirtied), it is like any other page. Dirty pages are copied back and forth between memory and a special ***swap file***.

Dynamic Memory Allocation

Programmers use ***dynamic memory allocators*** (such as ***malloc, free, realloc, and calloc***) to acquire VM at run time. Dynamic memory allocators manage an area of process virtual memory known as the ***heap***. The allocator maintains the heap as collection of variable-sized ***blocks***, which are either ***allocated*** or ***free***. ***Explicit allocators*** are used to allocate and frees space (e.g., malloc and free in C). ***Implicit allocators*** are used to allocate, but not free space (e.g., garbage collection in Java or Python).

#include <stdlib.h>

void *malloc(size_t size)

- Successful:
  - Returns a pointer to a memory block of at least **size** bytes aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
  - If **size == 0**, returns NULL
- Unsuccessful: returns NULL (0) and sets **errno**

void free(void *p)

- Returns the block pointed at by **p** to pool of available memory
- **p** must come from a previous call to **malloc** or **realloc**

Other functions

- **calloc**: Version of **malloc** that initializes allocated block to zero.
- **realloc**: Changes the size of a previously allocated block.
- **sbrk**: Used internally by allocators to grow or shrink the heap

- **void *realloc(void *ptr, size_t size);**

    - **ptr** → pointer to the memory area to be reallocated
    - **size** → new size of the array in bytes
- **void *calloc(size_t nitems, size_t size);**
    - **nitems** → number of elements to be allocated
    - **size** → size of elements

There are two important performance goals: maximize throughput and maximize peak memory utilization, which are often conflicting goals. ***Throughput*** is the number of completed requests per unit time. ***Peak Memory Utilization (U_k)*** is the max ***aggregate payload (P_k)*** divided by the ***current heap size (H_k)***. Note that the heap only grows when the allocator uses sbrk. The aggregate payload is the sum of currently allocated payloads that resulted from malloc. Poor memory utilization is caused by ***fragmentation***. For a given block, ***internal fragmentation*** occurs if payload is smaller than block size, whereas ***external fragmentation*** occurs when there is enough aggregate heap memory, but no single free block is large enough.

There are five major design issues:

1. Marking blocks as allocated
2. Picking a free block
3. Re-insert free block
4. Keep track of free blocks
5. Knowing how much to free given only a pointer

Alongside five solutions:

1. ***"Standard"*** → keep length of a block in the word preceding block (header)
2. ***Implicit List*** → use length of all blocks to "link" all blocks
3. ***Explicit List*** → use pointers to link free blocks

4. ***Segregated Free List*** → different lists for different size classes
5. ***Blocks Sorted by Size*** → use blocks sorted by size (balanced tree w/ length as key)

First fit searches list from beginning, choose first free block that fits. It can take linear time in total number of blocks (allocated and free), though it can cause "splinters" in practice. ***Next fit*** is like first fit, but the search list starts where previous search finished. However, some research suggests that fragmentation is worse. ***Best fit*** searches the list, chooses the best free block (fits, with fewest bytes left over) and keeps fragments small which usually improves memory utilization. You can allocate in a free block via ***splitting***, since the allocated space might be smaller than free space, we might want to split the block. Or you can join (***coalesce***) with next/previous blocks if they are free. The coalesce backwards, we use ***bidirectional coalescing*** with ***boundary tags*** that replicate size/allocated word at end of the blocks. This allows us to traverse the "list" backwards but requires extra space. However, boundary tags can cause internal fragmentation.

Garbage collection is the automatic reclamation of heap-allocated storage. The application never has to free. It is common in many dynamic languages (i.e., Python, Ruby, Java, JavaScript, Perl, ML, Lisp, Mathematica). There are variants ("conservative" garbage collectors) that exist for C and C++; however, they cannot necessarily collect all garbage. ***Mark-and-sweep collection*** does not move blocks (unless you also "compact"). Allocate using malloc until you "run out of space". When out of space, use extra ***mark bit*** in the

head of each block. ***Mark*** starts at roots and set mark bit on each reachable block. ***Sweep*** scans all blocks and free blocks that are not marked.

We view memory as a directed graph. Each block is a node in the graph and each pointer is an edge in the graph. Locations not in the heap that contain pointers into the heap are called ***root*** nodes (e.g., registers, locations on the stack, global variables). A node (block) is ***reachable*** if there is a path from any root to that node. Non-reachable nodes are ***garbage*** (cannot be needed by the application).

Files & Directories

The three fundamental abstractions:

1. Processes/Threads ← OS → CPU
2. Address Space ← OS → Memory
3. Files ← OS → Devices (Disk)

A ***file*** is a logical unit of data on a storage device, which can contain programs or data. Files can be structured or unstructured (e.g., UNIX implements files as a sequence of bytes (unstructured)). There are six attributes (additional information) that files store: Name, Type, Location, Size, Permissions, Creation Time. Furthermore, the major data operations are:

1. **create(name)** → Creating a File
    a. Allocate necessary disk space (checks quotas, permissions, etc.)
    b. Create a file descriptor (includes name, location, and other attributes)
    c. Add the file descriptor to the directory containing the file
    d. Optionally add file type / other attributes
2. **delete(name)** → Deleting a File

a. Find the directory containing the file
b. Free the disk blocks used by the file
c. Remove the ***<name, file descriptor>*** from the directory
d. Behavior is dependent on hard links

3. **open(name, mode) & close(fileID)** → Open/Close a File
   a. File permissions are checked
   b. Reference counts get changed
   c. File tables are updated
4. **read(fileID, from, size, buf) OR read(fileID, size, buf)** → Read a File
   a. OS reads "size" bytes from file position "from" into "buf"

```
for (i = from; i < from + size; i++)
        buf[i - from] = file[i];
```

   b. OS reads "size" bytes from current file position fp into buf, then advances fp

```
for (i = 0; i < size; i++){ buf[i] = file[fp + i];}
                fp += size;
```

5. **Write** → Write to a File
   a. operations are similar to reads, but copies the other direction
6. **Seek** → Seeks for a File
   a. just updates fp stored within the file table
7. ***Memory mapping*** a file
   a. Maps part of the virtual address space to a file
   b. Read/write operations to that portion of memory implies OS read/write from corresponding location in the file
   c. File accesses are simplified (no read/write calls are necessary)

File Access Methods (Programmer's View):

- ***Sequential*** → data processed in order
  - Most programs use this method
  - Ex: compiler reading a source file
- ***Direct*** → address a block based on a key value
  - Databases often do this
  - Ex: hash table, dictionary

File Access Methods (Operating System's View):

- ***Sequential*** → keep a pointer to the next byte in the file; update pointer on each read/write operation
- ***Direct*** → address any block in the file directly provided its offset within the file

Operating systems need a way to refer to files that are stored on disk, as they use numbers for each file. A ***directory*** is an OS data structure to map names to file descriptors (since users prefer textual names to refer to files). There are three naming strategies. The first is a ***single-level directory***, which uses one namespace for the entire space, every name unique and uses a special area of disk to hold the directory (containing <name, fileDesc> pairs). The next is a ***two-level directory***, where each user has a separate directory, but each and every user's files must still have unique names. Finally, there is a ***multi-level directory*** which is a tree-structure namespace (UNIX and all modern OSes). It stores directories on disk, just like files, except that the type field in file descriptor indicates it's a directory. User programs read directories just like any other file, but only special system calls can write directories (***open, creat, link, mkdir, rmdir***). Each directory contains ***<name, fileDesc>*** pairs in arbitrary order. The file referred to by name could be

another directory. There is one special "root" directory (/).

There are two kinds of links in directories. A ***hard link*** **(UNIX: ln command)** adds a second (or more) connection to a file. The OS maintains reference counts and deletes only when count == 0. The other is a ***soft link*** **(UNIX: ln -s command)** only makes a symbolic pointer from one file to another. Removing "A" leaves the name B in the directory, but it's a dangling pointer.

The core directory operations are:

- ***Search*** → locate an entry for a file in the current directory
- ***Create*** → adds a directory entry
- ***Delete*** → removes a directory entry
- ***List*** → list all files (**ls** command in UNIX)
- ***Rename*** → renames a file (**mv** command in UNIX)
- ***Traverse*** → traverse the directory

The OS must allow users to control sharing of files by granting/denying access to file operations based on protection info. ***Access Lists/Groups (Windows NT)*** keeps an access list for each file with username and type of access. Whereas ***Access Control Bits (UNIX)*** has three categories of user (**o**wner, **g**roup, **w**orld), and three types of access privileges (**r**ead, **w**rite, e**x**ecute). It maintains a bit for each combination: 111 101 000 → rwx r-x ---. Linux maintains all these permissions by admin users managing all the passwords. If you want to change your access, one must have admin rights for running commands on the terminal and carry out different tasks.

Physical Disks

The ***Hard Disk Drive (HDD)*** consists of ***platters***, each with two ***surfaces***. Each surface consists of concentric rings called ***tracks***. Each track consists of ***sectors*** separated by ***gaps***. The aligned tracks form a ***cylinder*** around the ***spindle***.

The ***disk capacity*** is the maximum number of bytes that can be stored (usually expressed in gigabytes (GB) or terabytes (TB), where 1 GB = 109 Bytes and 1 TB = 1012 Bytes). The capacity is determined by these technology factors:

- ***Recording density (bits/in):*** number of bits that can be squeezed into a 1-inch segment of a track
- ***Track density (tracks/in):*** number of tracks that can be squeezed into a 1-inch radial segment
- ***Areal density (bits/in^2):*** product of recording and track density

$$\text{Capacity} = (\#\text{ bytes/sector}) \times (\text{avg. } \#\text{ sectors/track}) \times (\#\text{ tracks/surface}) \times (\#\text{ surfaces/platter}) \times (\#\text{ platters/disk})$$

The disk surface spins at a fixed rotational rate. The read/write head is attached to the end of the arm and flies over the disk surface on a thin cushion of air. By moving radially, the arm can position the read/write head over any track.

The ***Solid-State Drive (SSD)*** has faster sequential access than random access, but the random writes are somewhat slower. Erasing a block takes a long time (~1 ms) and modifying a block page requires all other pages to be copied to new block. In earlier SSDs, the

read/write gap was much larger. This was fixed by flash translation layer. Data is read/written in units of pages. A page can be written only after its block has been erased. A block wears out after 100,000 repeated writes. Pages range from 512 B to 4 KB, and blocks range from 32 to 128 pages.

The average time to access some target sector approximated by: $T_{access} = T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer}$

1. ***Seek Time*** ($T_{avg\ seek}$)
   a. Time to position heads over cylinder containing target sector
   b. Typical $T_{avg\ seek}$ 3 – 9 ms
2. ***Rotational Latency*** ($T_{avg\ rotation}$)
   a. Time waiting for first bit of target sector to pass under r/w head
   b. Most costly
   c. $T_{avg\ rotation}$ = 1/2 x 1/RPMs x 60 sec/1 min
   d. Typical $T_{avg\ rotation}$ = 7,200 RPMs
3. ***Transfer Time*** ($T_{avg\ transfer}$)
   a. Time to read the bits in the target sector
   b. $T_{avg\ transfer}$ = 1/RPM x 1/(avg # sectors/track) x 60 secs/1 min

Modern disks present a simpler abstract view of the complex sector geometry, as the set of available sectors is modeled as a sequence of B ***logical blocks*** (0, 1, 2, ..., B - 1). Mapping between logical blocks and actual (physical) sectors is maintained by hardware/firmware device called a disk controller. This converts requests for logical blocks into (surface/head, track, sector) triples, and allows the controller to set aside spare cylinders for each zone. This accounts for the difference in "formatted capacity" and "maximum capacity".

Disk head scheduling permutes the order of disk requests from the order they arrive to minimize number or length of seeks to perform. This is implemented in one of four ways:

1. ***First-Come, First-Serve (FCFS)***
    a. No overhead, works well for light loads, also for SSDs
    b. Handles I/O requests sequentially
    c. Fair to all processes
    d. Approaches random scheduling in performance if there are many requests
    e. Suffers from global zigzag effect
2. ***Shortest Seek Time First (SSTF) / Shortest Seek Distance First (SSDF)***
    a. Always go to next closest request, maintain doubly linked sorted list
    b. Selects the request with the minimum seek time from the current head position
    c. Easier to compute distances
    d. It's biased in favor of the closest cylinder requests
    e. Is a form of SJF scheduling, may cause starvation of some requests
3. ***SCAN***
    a. Service requests as we pass them
    b. Requires sorted list of requests
    c. Disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed, and servicing continues
    d. It moves in both directions until both ends
    e. Tends to stay more at the ends so fairer to the extreme cylinder requests

4. ***Circular Scan (C-SCAN)***
    a. Like SCAN, but once it hits the end, it jumps to the other end and moves in the same direction
    b. The head moves from one end of the disk to the other, servicing requests as it goes. When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.
    c. Treats the cylinders as a circular list that wraps around from the last cylinder to the first one
    d. Provides a more uniform wait time than SCAN; it treats all cylinders in the same manner

File Organization

The ***file descriptor*** (***inode*** in Linux) is a structure used to describe where the file is on the disk. It also maintains a list of attributes of the file and must be stored on the disk just like all other files. Most systems have a majority of small files, have the majority of disk space taken up by large files, and have I/O operations target both small and large files. The per-file cost must be low, but large files must also have good performance.

There are four major implementations of file organization:

1. ***Contiguous Allocation (Early PCs)***
    a. (Pro) Good performance
    b. (Con) Changing file sizes
    c. (Con) Fragmentation and trying to find chunks of sufficient size
2. ***Linked Files (MS-DOS)***

a. Keep a list of all the free blocks. In the file descriptor, keep a pointer to the first block. In each block, keep a pointer to the next block.
b. (Pro) Fragmentation
c. (Pro) File size changes
d. (Pro) Efficient support of sequential access
e. (Con) Does not support direct access
f. (Con) Seek for every block

3. Indexed Files (Hash Tables & B-Trees)

a. File descriptor has an array of block pointers for each file. The user (or OS) must declare maximum length of the file when created, where the OS then allocates an array to hold pointers to all blocks (but only allocates blocks on demand). The OS then fills in the pointers as it allocates blocks.
b. (Pro) Minimal wasted space for files (doesn't allocate until it needs to)
c. (Pro) Direct access for each block
d. (Pro) Sequential access is also efficient
e. (Con) Wasted space in FDs
f. (Con) Limited file size supported due to amount of blocks
g. (Con) Lots of seeks due to non-contiguous data

4. Multi-Level Indexed Files (Linux)

a. Each FD contains k block pointers. The first k-2 block pointers point to data. The k-1st pointer points to a block of 2^d pointers to 2^d more data blocks. The k^{th} pointer points to a block of pointers to indirect blocks.
b. (Pro) Incremental file growth
c. (Pro) Small and large files
d. (Con) Indirect access is inefficient for Direct
e. Access

f. (Con) Lots of seeks because non-contiguous data

Free space management is a way to keep track of which disk blocks are free, and to be able to find free space quickly and release space quickly. The solution is a ***bitmap***, which has one bit for each block on the disk. If the bit is 0, the block is free, else the bit is 1 and block is allocated. We can quickly determine if any page in the next 32 is free (32-bit) by comparing to "word" 0. Marking a block as freed is as simple as toggling a bit.

However, the bitmap might be too big to keep in memory for a large disk. If most of the disk is in use, it will be expensive to find free blocks with a bitmap. • An alternative implementation is to link together free blocks, known as a ***free list***. The head of the list is cached in kernel memory, with each block containing a pointer to the next free block.

Threads

A ***thread*** of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.

Similarities between a process and a thread:

- Each has its own logical control flow
- Each can run concurrently with others (possibly on different cores)
- Each is context switched

Differences between a process and a thread:

- Threads share all code and data (except local stacks), but processes (typically) do not

- Threads are somewhat less expensive than processes, whereas process control (creating and reaping) twice as expensive as thread control

Pthread Library:

- ***pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr, void *(*start_routine)(void *), void *restrict arg);***
  - Starts a new thread in the calling process.
  - **thread** → the location where the ID of the newly created thread should be stored, or NULL if the thread ID is not required
  - **attr** → the thread attribute object specifying the attributes for the thread that is being created, or if attr is NULL, the thread is created with default attributes
  - **start_routine** → the main function for the thread; the thread begins executing user code at this address
  - **arg** → argument passed to start_routine
- ***pthread_join(pthread_t thread, void **retval);***
  - waits for the thread specified by thread to terminate
  - If that thread has already terminated, then pthread_join() returns immediately
  - If retval is not NULL, then pthread_join() copies the exit status of the target thread into the location pointed to by retval
  - If the target thread was canceled, then PTHREAD_CANCELED is placed in the location pointed to by retval
- ***pthread_self(void);***
  - Returns the ID of the calling thread
- ***pthread_cancel(pthread_t thread);***

    - Sends a cancellation request to the given thread
- ***pthread_exit(void *retval);***
    - Terminates the calling thread and returns a value via retval that (if the thread is joinable) is available to another thread in the same process that calls pthread_join(3)

Data Sharing

Global variables are variables that are declared outside of a function. Virtual memory contains exactly one instance of any global variable. ***Local variables*** are variables declared inside function without a static attribute. Each thread stack contains one instance of each local variable. ***Local static variables*** are variables declared inside function with the static attribute. Virtual memory contains exactly one instance of any local static variable.

Shared data is data that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies. The ***critical section*** refers to the segment of code where processes access shared resources, such as common variables and files, and perform write operations on them. ***Mutual exclusion*** states that "no two processes can exist in the critical section at any given point of time".

A ***progress graph*** depicts the discrete execution state space of concurrent threads. Each axis corresponds to the sequential order of instructions in a thread. Each point corresponds to a possible execution state. A ***trajectory*** is a sequence of legal state transitions that describes one possible concurrent execution of the

threads. Instructions in critical sections should not be interleaved. Sets of states where such interleaving occurs form ***unsafe regions***. A trajectory is safe if and only if it does not enter any unsafe region. A trajectory is correct if and only if it is safe.

A ***semaphore*** is a non-negative global integer synchronization variable, which is manipulated by P and V operations. A ***binary semaphore*** is a semaphore whose value is always 0 or 1. A ***mutex*** is a binary semaphore that is used for mutual exclusion. The ***P*** operation locks the mutex, whereas the ***V*** operation unlocks the mutex. There is also ***"holding"*** a mutex, which means it is locked and not yet unlocked. A ***counting semaphore*** is used as a counter for a set of available resources.

The ***Bounded Buffer Problem*** (a.k.a. Producer-Consumer Problem) occurs when there is a buffer of n slots, and each slot can store one unit of data. There are two processes running, namely, producer and consumer, which are operating on the buffer. A producer tries to insert data into an empty slot of the buffer while a consumer tries to remove data from a filled slot in the buffer concurrently.

The solution is to use semaphores (one mutex and two counting semaphores) to prevent these processes from running concurrently. The mutex enforces mutually exclusive access to the buffer, one counting semaphore is used to count the available slots in the buffer, and the other counting semaphore counts the available items in the buffer.

The ***Readers-Writers Problem*** is a generalization of the mutual exclusion problem. The reader threads only read

the object, and the writer threads modify the object. The writers must have exclusive access to the object, with an unlimited number of readers having access to the object.

There are two possible solutions. The first states that no reader should be kept waiting unless a writer has already been granted permission to use the object. A reader that arrives after a waiting writer gets priority over the writer. The second states that once a writer is ready to write, it performs its write as soon as possible. A reader that arrives after a writer must wait, even if the writer is also waiting. ***Starvation*** (where a thread waits indefinitely) is possible in both cases.

Functions called from a thread must be thread-safe. A function is ***thread-safe*** if and only if it will always produce correct results when called repeatedly from multiple concurrent threads. There are fours classes of ***thread-unsafe*** functions:

1. Functions that do not protect shared variables
2. Functions that keep state across multiple invocations
3. Functions that return a pointer to a static variable
4. Functions that call thread-unsafe functions

A ***race*** occurs when correctness of the program depends on one thread reaching point x before another thread reaches point y. A process is ***deadlocked*** if and only if it is waiting for a condition that will never be true.