CS2106 Final Cheatsheet

@Wxy2003-xy

- The child process from `fork()` receives a copy-on-write duplicate of the parent's memory and file descriptor table.
- Use `wait()` or `waitpid()` in the parent to reap the child and avoid zombie processes.
- Each child should call `exit()` (or return from `main()`) to terminate cleanly.
- `exec*()` functions replace the current process image with a new program. No return, no return back to the original prog

Function	Arguments	PATH	CustEnv
`execl(path, arg0, ..., NULL)`	Arg list, full path	No	No
`execlp(file, arg0, ..., NULL)`	Arg list, search `PATH`	Yes	No
`execle(path, arg0, ..., NULL, envp)`	Arg list, full path	No	Yes
`execv(path, argv[])`	Arg vector, full path	No	No
`execvp(file, argv[])`	Arg vector, search `PATH`	Yes	No
`execvpe(file, argv[], envp[])`	Arg vector, `PATH`, env	Yes	Yes

- `int id=shmget(IPC_PRIVATE, size, IPC_CREAT|0600);`
- `alloc = (T*) shmat(id, NULL, 0);`
- `shmdt(alloc)`
- `shmctl(id, IPC_RMID, 0);`
- `pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)`
  Spawns a thread that shares the same address space. Returns 0 on success.
- `pthread_join(pthread_t thread, void **retval)`
  Waits for a thread to finish and optionally retrieves its return value. Returns 0 on success.
- `pthread_exit(void *retval)`
  Terminates the calling thread and makes `retval` available to `pthread_join()`. Does not return.
- `sem_init(sem, [0(thread)|1(process)], value)` — Initialize semaphore to given value.
- `sem_wait(sem)` — Decrement; blocks if value is 0.
- `sem_post(sem)` — Increment; unblocks one waiter if any.
- `sem_destroy(sem)` — Cleans up; does not free memory.
- Used for mutual exclusion (binary semaphores) or limiting access (counting semaphores).

Dining Philosophers: Limit-Seat Strategy

- Limits the number of philosophers who can attempt to pick up chopsticks to ensure progress.
- Prevents circular wait, breaking one of Coffman's deadlock conditions.
- Starvation is still possible due to unfair scheduling.

```
typedef struct {
    int _status[N];
    sem_t mutex;
    sem_t sem[N];
} SharedMem;
void takeChpStk(SharedMem* shm, int i)
    sem_wait(&shm->mutex);
    shm->_status[i] = HUNGRY;
    safeToEat(shm, i);
    sem_post(&shm->mutex);
    sem_wait(&shm->sem[i]);
void safeToEat(SharedMem* shm, int i)
    if ((shm->_status[i] == HUNGRY) &&
        (shm->_status[LEFT] != EATING) &&
        (shm->_status[RIGHT] != EATING)) {
            shm->_status[i] = EATING;
            sem_post(&shm->sem[i]);}
void putChpStk(SharedMem* shm, int i)
    sem_wait(&shm->mutex);
    shm->_status[i] = THINKING;
    safeToEat(shm, LEFT);
    safeToEat(shm, RIGHT);
    sem_post(&shm->mutex);
```

Barrier turnstile

```
int *count;
sem_t *barrier, *mutex, *barrier2;
void init_barrier(int numproc) {
  sem_init(barrier, 1, 0);
  sem_init(barrier2, 1, 1);
  sem_init(mutex, 1, 1);
  *count = 0;
}
void reach_barrier() {
  sem_wait(mutex); //==================================↓
  (*count)++;
  if (*count == nproc) {
    sem_wait(barrier2); // ----------------------------↓
    sem_post(barrier);} // - - - - - - - - - - - - - - ↑
  sem_post(mutex); //==================================↑
  sem_wait(barrier); // - - - - - - - - - - - - - - - -↓
  sem_post(barrier); // - - - - - - - - - - - - - - - -↑
  sem_wait(mutex); //==================================↓
  (*count)--;
  if (*count == 0) {
    sem_wait(barrier); // - - - - - - - - - - - - - - -↓
    sem_post(barrier2);} // ---------------------------↑
  sem_post(mutex); //==================================↑
  sem_wait(barrier2); // ----------------------------↓
  sem_post(barrier2); // ----------------------------↑
}
```

- **Turnaround t**: Total time from job arrival to completion.
- **Response t**: Time from job arrival to first CPU execution.
- **Waiting time**: Time a job spends in the ready queue.
- **Throughput**: Number of jobs completed per unit time.

Algorithm	Preemptive	Fairness	Resp T	Turarnd T	Starv
FCFS	No	by order	↓	↑ (convoy effect)	↓
SJF	No	No	↑ short jobs	Optm (theo)	↑
SRT	Yes	No	↑ short jobs	Optm	↑
RR	Yes	Yes	↑	dep (qtm)	Low
Lottery	Can		Fair	Fair	↓
MLFQ	Yes	Adaptive	↑	Adaptive	dep

Table storage:

- UPP: user process pages
- SWAP: Non-memory resident iser process page
- Process page table: in PCB table in OS mem region in RAM
- Open file table: in OS mem region in RAM
- File descriptor table: in PCB
- Dynamically allocated mem in a prgram: UPP or SWAP
- file decriptor returned from an open(...) syscall: UPP or SWAP
- compiled binary files: not part of the virtual mem

Contiguous mem:

- **Tracking free space:**
  - Bitmap: 1 bit per block, where 0 = free, 1 = allocated.
  - Linked List: `<status, start_idx, size, next*>`
  - Buddy System: 2^n sized buddies: ad_a XOR $ad_b = 2^n$
  - **the rightmost bit is bit0, also the LSB**
- **Fragmentation:**
  - Internal (never in contiguous alloc if carve perfectly)
  - External: Gaps between allocated blocks.

Paging

- Fixed-size units: Logical pages and physical frames.
- Page Table: Maps pages to frames.
- TLB: Hardware cache for recent page table entries.

Segmentation

- Logical memory divided into named segments (text, data, stack, heap).
- Each has a base and limit.
- Logical Address = `<Segment_ID, Offset>`.
- Different segments don't share page. As text and data can have different permissions

Virtual mem

- Logical memory can exceed physical memory.
- $|\text{Virtual pages}| = 2^{\text{Virtual\_address bits}}$

Demand Paging

- Pages are only loaded on access. No memory resident page
- (+) Reduces startup time and memory usage.
- (-) more page fault at start; page fault can cascade on other processes (e.g. thrashing)

Page Access

```
Check page table:
    if memory resident: acess physical mem; done;
    else: [page fault] -> trap to OS
        locate page in secondary storage;
        load into physical mem;
        update page table;
        goto Check page table;
```

$$\bar{T}_{access} = p_{TLB\_hit}(T_{TLB} + T_{RAM}) + (1 - p_{TLB\_hit}) \cdot [T_{RAM} + p_{page\_fault} T_{pf\_remedy} + (1 - p_{pf}) T_{RAM}]$$

Single-Level Direct paging

- Wasteful for sparse address spaces.

- page table size = $|page| \times$ `sizeof(entry)`

Multilevel
- Page directories point to page tables. dir size = page size
- max level:
  1. find branching factor `bf = page_size / entry_size`
  2. virtual address bits: `[(level × branching):ofst]`
  3. min mem for page table: `level × entry_size`
  4. max: `[(level - 1) × frames + 1] × entry_size`
- page dir base reg →<`page_dir#, page#, ofst`>
- overhead = `sizeof(page_dir) +` $\sum$`sizeof(loaded leaf table)`
- min: clustered pages; max: sparse pages(more leaf tables loaded)

Inverted Page Table
- One entry per frame: `<pid, page#>` → frame.
- Compact but slow due to full-table lookup.
- page# is not unique among processes
- pid + page# can uniquely identify a memory page
- (+) Efficient: One table for all processes;(-) Slow translation

Temporal Locality: Recently used memory will be used again.
Spatial Locality: Nearby memory addresses.

Page Replacement Algorithms
- When a page is evicted
  - Clean page: not modified → no need to write back
  - Dirty page: modified → need to write back
- OPT: Replace page with furthest next use (ideal).
- FIFO: Oldest page out. (temporal locality → Belady's Anomaly: more frames, more page faults)
- LRU: Least recently used page.
- Clock: Approximate LRU using reference bits.

```
// Modified FIFO to give a second chance to pages
     that are accessed
// Each PTE now maintains a "reference bit"
// 1 = Accessed, 0 = Not accessed
while (The oldest FIFO page is selected):
  If reference bit == 0
    Page is replaced
  If reference bit == 1
    Page is given a 2nd chance
    Reference bit cleared to 0
    Next FIFO page is selected
// When all pages have reference bit == 1
// Degenerates into FIFO algorithm
```

Frame allocation: N frames, M processes
- Equal alloc: Each process get N / M frames
- Proportional alloc: P gets $N \frac{\text{size}_P}{\sum \text{size}_{P_i}}$

Local Replacement
- Only evict pages from the same process.
- Predictable and isolated.
- if not enf allocated, hinders process progress

Global Replacement
- Victim page can belong to any process.
- More flexible, allows self-adjustment, but less stable.
- bad behaved process can affect others

Thrashing
- Excessive page faults reduce CPU utilization.
- Can lead to cascading faults in global replacement.
- **Working Set Model:** Allocate enf frames for `W(t, Δ)`.

Aspect	Memory Management
Underlying Storage	RAM
Access Speed	Constant
Unit of Addressing	Physical memory address
Usage	Address space for process **Implicit** when process runs
Organization	**Paging/Segmentation**: determined by HW & OS

Aspect	File System Management
Underlying Storage	Disk
Access Speed	Variable disk I/O time
Unit of Addressing	Disk sector
Usage	Non-volatile data **Explicit** access
Organization	**Many FS types**: ext* (Linux), FAT* (Windows), HFS* (Mac)

- A file is the smallest amount of information that can be written to secondary memory. It is a named collection of data, used for organizing secondary memory
- A file type is a description of the information contained in the file. A file extension is a part of the file name that follows a dot and identifies the file type
- `fork()` and `open(`f`)` → `ref_count`$_f$`++`
- `Close(`f`)` → `ref_count`$_f$`--`.
- Truncating a file means that all the information on the file is erased but the administrative entries remain in the file tables. Occasionally, the truncate operation removes the information from the file pointer to the end.

- **Protection**: [user, grp, others] `chmod 754` → `drwxr-xr--`
- A process uses the `open()` system call to access a file:
- Updates the process's **Per-Process File Descriptor Table**: `fd` points to the corresponding system-wide table entry.
- Shared file descriptors:
  - Two `fd`s pointing to the same system-wide entry share offset and metadata.
  - Created using `dup/2()`, or inherited from `fork()`.
  - Reading by one process advances the file offset, affecting the other process.
  - call `open()` separately or `dup()` to not share ofst

FAT12	FAT16	FAT32
2^{12} clusters	2^{16}	2^{28}

- `open(const char *pathname, int flags[,mode])`:
  - Opens a file and returns a file descriptor (int).
  - `flags`: `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT`, etc.
  - `mode` required if `O_CREAT` is used, set permission bits.
- `read(int fd, void *buf, size_t count)`:
  - Reads up to `count` bytes from `fd` into buffer `buf`.
  - Returns the number of bytes read, or 0 on EOF.
  - Advances the file offset by the number of bytes read.
- `write(int fd, const void *buf, size_t count)`:
  - Writes up to `count` bytes from buffer `buf` to `fd`.
  - Returns the number of bytes written.
  - Advances the file offset by the number of bytes written.
- `lseek(int fd, off_t offset, int whence)`:
  - Moves the file offset for `fd`.
  - `whence` can be `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`.
- `close(int fd)`:
  - Closes `fd`.
  - Releases the open file table entry.

File block allocation:
- Contiguous: (+) simple and fast; (-) ext.frag, files size need to be specified first
- Linkedlist: (+) no frag; (-) slow rand. access, extra mem usage for pointers, unreliable
- Linkedlist2: (+) faster (in mem traverse); (-) FAT keep tracks of all disk blocks in a partition, wasteful
- Indexed: IndexBlock[N] == Nth Block address. (+) fast, Only index block of opened file needs to be in memory; (-) Limited maximum file size

Directory structure:
- Linkedlist: Each entry represents a file; Store file name (minimum) and possibly other metadata; Store file information or pointer to file information
- Hashtable: HashTable[K] is inspected to match file name. Usually chained collision resolution is used
- File info: File information consists of: File name and other metadata; Disk blocks information

- EXT2 inode max file size:
- x_0 direct, x_1 single-level indirect... x_n n-level indir
- t data block size, p pointer size
- inode size: $p \sum_{i=0}^{n} x_i$
- max file size: $t \sum_{i=0}^{n} x_i (\frac{t}{p})^i$

Open:

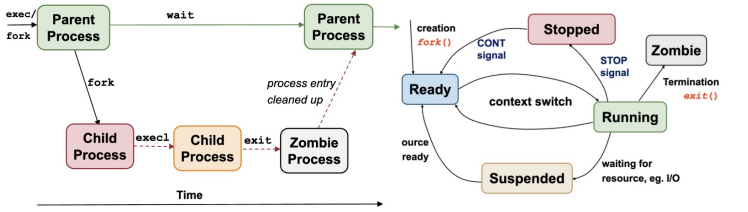
```
Process P opens file /.../.../.../F:
  Search system-wide table for existing entry E
    If found:
      Creates an entry in P's table to point to E
      Return a pointer to this entry
    If not found, continue;
  Use full pathname to locate file F
    If not found, open operation terminates with error
    When F is located:
      its file information is loaded into a new entry E in
          system-wide table;
      Creates an entry in P's table to point to E
      Return a pointer to this entry
// The returned pointer is used for further read/write
     operation
```

- A = 1010, B = 1011, C = 1100, D = 1101, E = 1110

Feature	Contiguous	Linked List	FAT	Inode-based (e.g., ext)
Access time (random)	Fast	Slow	Moderate	Fast
Access time (sequential)	Fast	Fast	Fast	Fast
Disk fragmentation	High	None	None	Low
Supports random access	Yes	No	Yes (with FAT table)	Yes
Space efficiency	Poor	Good	Good	Very good
Pointer overhead	None	High	High (FAT in memory)	Low (indirect blocks)
File size flexibility	Poor	Good	Good	Excellent
Crash recovery	Poor	Poor	Moderate	Good (journaling)

- 1 Kb = 1024 byte 1 Mb = 2^{20} byte 1 Gb = 2^{30} byte
- $2^8 = 256, 2^9 = 512, 2^{10} = 1024 = 1Kb, 2^{12} = 4096 = 4Kb$
- $2^{16} = 65,536 = 64Kb$, $2^{20} = 1,048,576 = 1Mb$
- $2^{32} = 4,294,967,296 = 4Gb$

- Load and Save racing condition:
- $\forall I_i \rightarrow \{I_{i,Load}, I_{i,Save}\}$
- All permutations of $\mathbb{I}$: $\{I_i | i \leq n\}$,
  where $\forall I_i \in \mathbb{I}, I_{i,Load} \prec I_{i,Save} \wedge I_{i,Save} \prec I_{i+1,Load}$

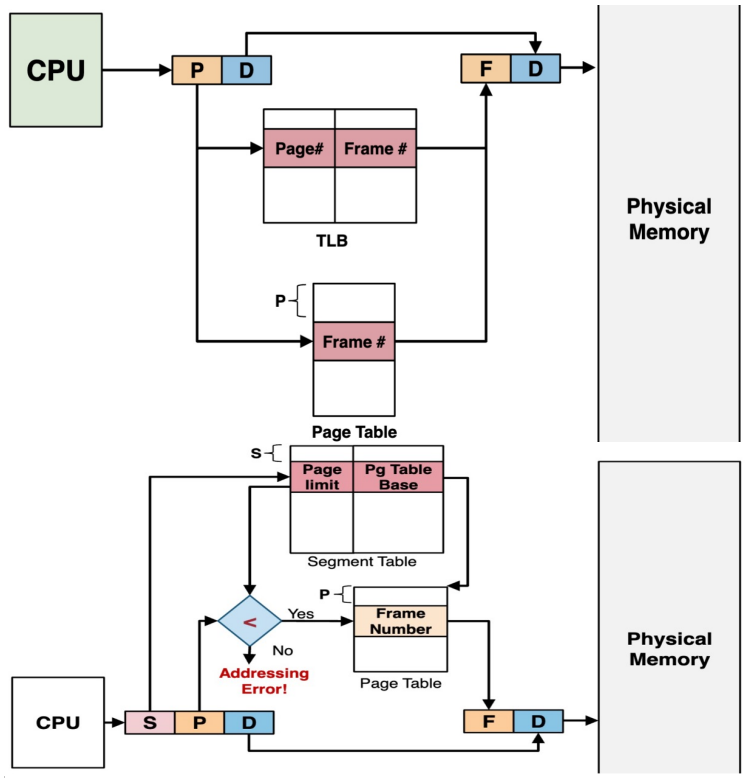


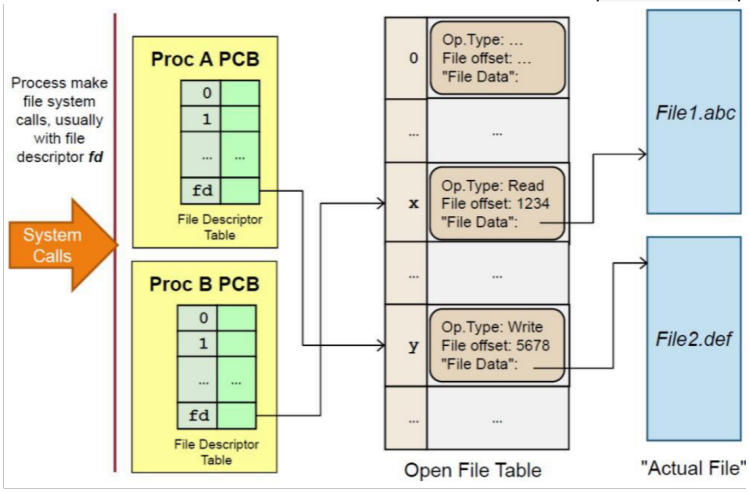

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void myOwnHandler( int signo )
{
    if (signo == SIGSEGV){
        printf("Memory access blows up!\n");
        exit(1);
    }
}

int main(){
    int *ip = NULL;

    if (signal(SIGSEGV, myOwnHandler) == SIG_ERR)
        printf("Failed to register handler\n");

    *ip = 123;

    return 0;
```

User defined function to handle signal. In this example, we handle the "SIGSEGV" signal, i.e. the memory segmentation fault signal.

Register our own code to replace the default handler.

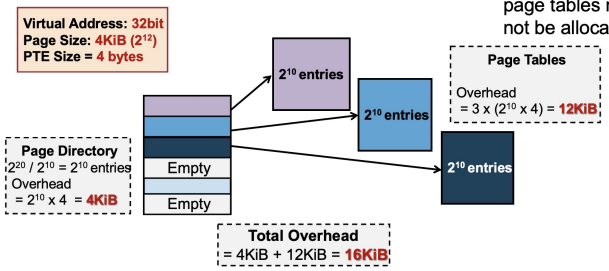
This statement will cause a segmentation fault.



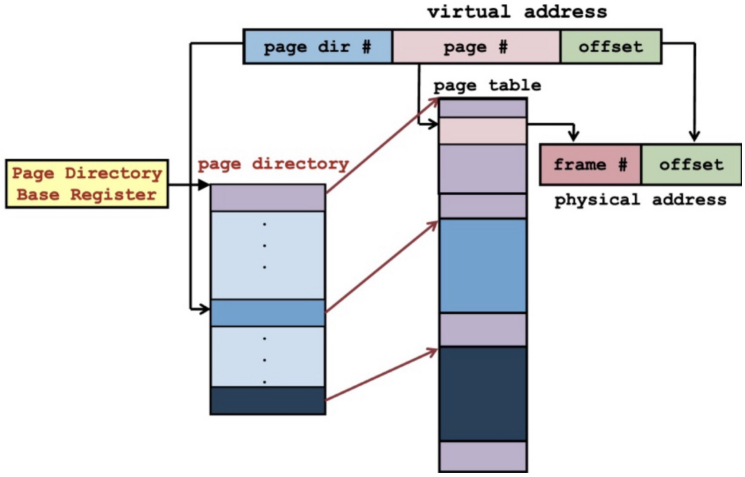


2-Level Paging: Advantage

- Using the same setting as the previous example
  - Assume only 3 page tables are in use
  - Overhead = 1 page directory + 3 smaller page tables

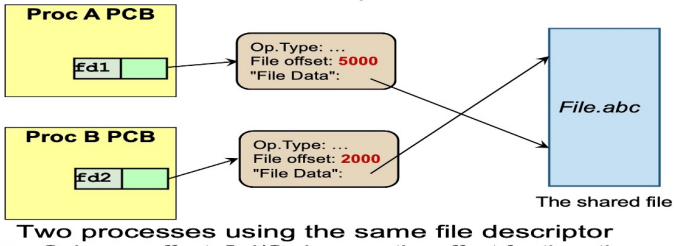


- We can have empty entries in the page directory
  - → The corresponding page tables need not be allocated!

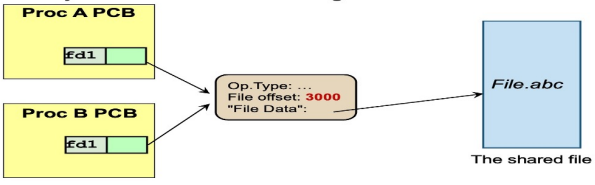


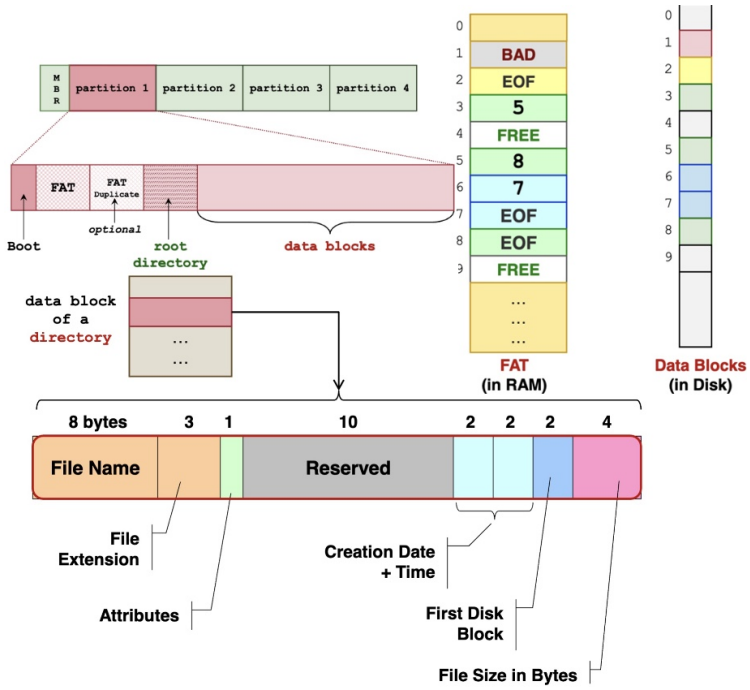
- **System-wide Open-File Table**
  - One entry per unique file
- **Per-Process Open-File Table**
  - One entry per file used in the process
  - Each entry points to the corresponding system-wide table entry

- Two processes using different file descriptors
  - I/O can occurs at independent offsets

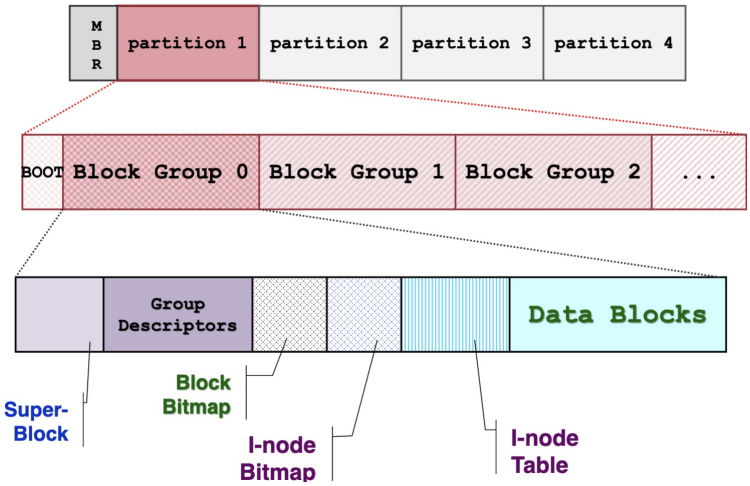


- Two processes using the same file descriptor
  - Only one offset → I/O changes the offset for the other process

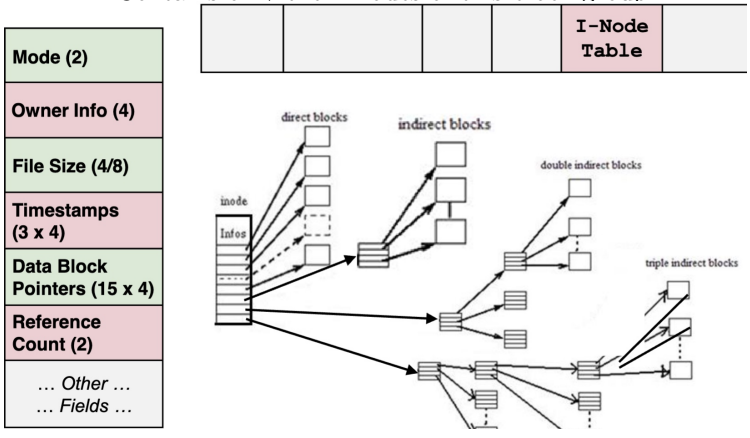




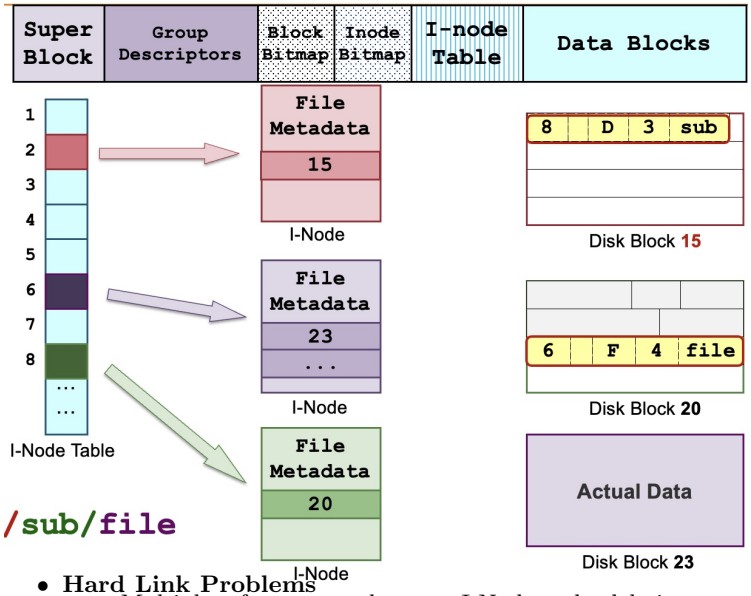
the actual size is a little lesser Special values (EOF, FREE, etc.) reduces total number of valid data block/cluster indices



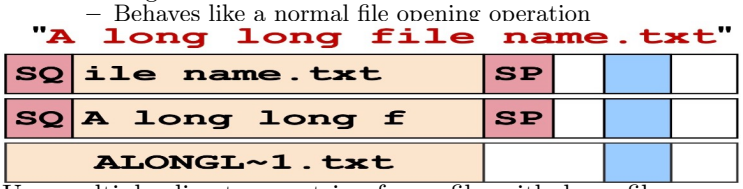
- **Superblock**
  - Describes the whole file system
  - Includes:
    - Total I-Nodes number, I-Nodes per group
    - Total disk blocks, Disk Blocks per group
    - etc.
  - Duplicated in each block group for redundancy
- **Group Descriptors**
  - Describe each of the block groups
  - Include:
    - Number of free disk blocks, free I-Nodes
    - Location of the bitmaps
  - Duplicated in each block group as well
- **Block Bitmap**
  - Tracks the usage status of blocks in this block group
  - 1 = Occupied, 0 = Free
- **I-Node Bitmap**
  - Tracks the usage status of I-Nodes in this block group
  - 1 = Occupied, 0 = Free
- **I-Node Table**
  - An array of I-Nodes
  - Each I-Node is accessed by a unique index
  - Contains only the I-Nodes of this block group



- **I-Node Contents**
  - Contains metadata for a file
  - Contains 15 disk block pointers (disk block numbers)
- **Direct Blocks**
  - First 12 pointers
  - Each points directly to a disk block containing actual data
- **Single Indirect Block**
  - 13th pointer
  - Points to a disk block containing a list of direct block pointers
- **Double Indirect Block**
  - 14th pointer
  - Points to a disk block that contains single indirect block pointers
- **Directory Data Blocks**
  - Store a linked list of directory entries
  - Each entry holds metadata for files or subdirectories contained in the directory
- **Each Directory Entry Contains:**
  - I-Node number for the file or subdirectory
  - Size of this directory entry (to locate the next entry)
  - Length of the file/subdirectory name
  - Type: File or Subdirectory
  - May also indicate other special file types
  - File/Subdirectory name (up to 255 characters)



- **Hard Link Problems**
  - Multiple references to the same I-Node make deletion management difficult
  - Requires maintaining an I-Node reference count
  - The reference count is decremented on each deletion
- **Symbolic Link Problems**
  - Stores only the pathname of the target file
  - Can be easily invalidated due to:
    - File name changes
    - File deletions
  - Requires a path lookup to resolve the actual I-Node of the target file
  - Behaves like a normal file opening operation



Use multiple directory entries for a file with long file name; Need to keep the 8+3 short version for backward compatibility

Unix Indexed Node