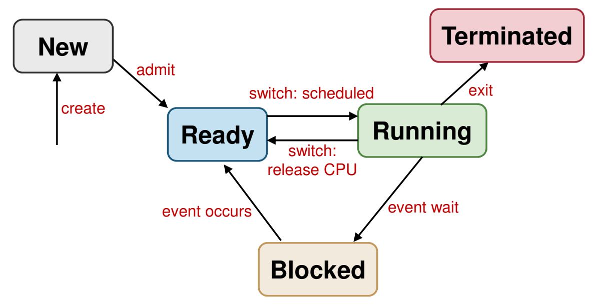
CS2106 Introduction to Operating Systems

Basics

- **Operating Systems ...**
  - Manages resources and coordination (process synchronization, resource sharing)
  - Simplify programming (abstraction of hardware, convenient services)
  - Enforce usage policies
  - Security of protection
  - User program portability (across different hardware)
  - Efficiency (optimized for particular usage and hardware)
- **Kernel mode**: complete access to all hardware resources
  User mode: limited access to hardware resources
- **Monolithic OS**: Kernel is one big special program
  Microkernel OS: Kernel is very small and clean, and only provide basic and essential facilities (e.g. IPC, address space management, thread management); higher-level services (device driver, process management, memory management, file system) built on top of basic facilities and runs outside the OS (using IPC to communicate). Kernel is more robust, and there is more isolation and protection between kernel and higher-level services, but at lower performance.
- **Type 1 Hypervisor**: Runs directly on hardware
  Type 2 Hypervisor: Runs on a host operating system

Process Abstraction

- **Generic 5-State Process Model**:



- **Process Control Block (PCB)**: information about a process: registers, memory region info, PID, process state
- **Syscall mechanism**:
  1. User program invokes library call
  2. Library call places the syscall number of designated location (e.g. register)
  3. Library call executes TRAP instruction to switch to kernel mode
  4. Appropriate system call handler is determined using the syscall number as index (usually handled by a dispatcher)
  5. Syscall handler is executed (this carries out the actual request)
  6. Control returned to library call, switched back to user mode
  7. Library call returns to user program
- **Exception**: Synchronous (occurs due to program execution, e.g. arithmetic errors, memory access errors). Executes an exception handler, like a forced syscall.
- **Interrupt**: Asynchronous (occurs independent of program execution, e.g. timer, mouse movement, keypress). Executes an interrupt handler, program execution is suspended.

Process Abstraction in Unix

- `pid_t fork(void)`:
  Parent returns PID of child, child returns zero
- `int execl(const char *path, const char *arg, ...)`
  `int execv(const char *path, char *const argv[])`
  e.g. `execl("/bin/ls", "ls", "-la", NULL)`
- `init` process is the root process (traditionally PID=1)
- `int wait(int *status)`
  Set `status` to `NULL` to ignore status
  Least significant 8 bits is the value passed to `exit(int)`

Process Scheduling

- **Non-preemptive (Cooperative)**: Process stays scheduled until it blocks or yields
  Preemptive: At the end of time quota, the process is suspended (it is still possible to block or yield early)

Batch Processing

No user interaction, non-preemptive scheduling is predominant

- **Turnaround time**: Total time taken from arrival to finish (including waiting time)
  Throughput: Number of tasks finished per unit time
  CPU utilization: Percentage of time CPU is doing work
- **First-come first-served**: Use FIFO queue based on arrival time (when tasked is blocked it is removed; it is placed at the back of queue when it is ready again). Guaranteed to have no starvation
  Shortest Job First: Select task with smallest CPU time (until next I/O). Starvation is possible because long job may never get a chance (when short jobs keep arriving). Prediction of CPU time usually uses exponential average of history
  Shortest Remaining Time: Preemptive version of SJF
- **Convoy effect**: Many tasks contend for CPU (while I/O is idle), and then contend for I/O (while CPU is idle)

Interactive Environment

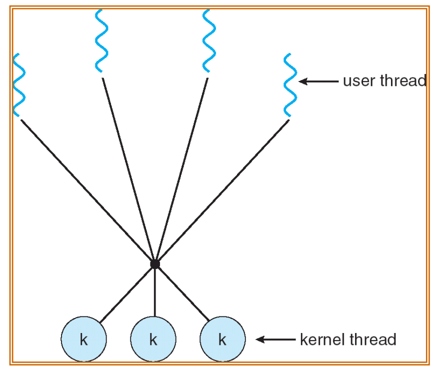
Preemptive scheduling algorithms are used to ensure good response time

- **Response time**: Time between request and response by system
  Predictability: Less variation in response time **Time quantum**: Execution duration given to a process, must be a multiple of timer interrupt
- **Round robin**: Like First-come first-served, but will be interrupted when time quantum elapses
  Priority scheduling: Each task gets a priority, highest priority gets scheduled first.
  Preemptive variant: new higher priority process preempts currently running lower priority process
  Non-preemptive variant: new higher priority process has to wait for next round of scheduling
  Low priority process can starve
  Priority inversion: higher priority task forced to block while lower priority task gets to run
  Multi-level feedback queue:
  If $Priority(A) > Priority(B)$ then A runs
  If $Priority(A) == Priority(B)$ then round-robin
  New job gets highest priority
  If a job fully utilized its time slice then priority reduced
  If a job yields/blocks then priority retained
  Lottery scheduling: Lottery tickets assigned to processes (possibly unevenly depending on priority), and randomly chosen winner is allowed to run (preemptive)
  Parent can distribute tickets to its child processes, and each shared resource (CPU, I/O) can have its own set of tickets

Threads

"Lightweight process"

- **Benefits**:
  Much less additional resources needed as compared to processes
  No need for additional mechanism to pass information between threads
  Multithreaded programs can appear much more responsive
  Multithread programs can take advantage of multiple CPUs
- **Problems**:
  Parallel syscall possible - have to guarantee correctness
  Process behaviour - `fork()/exec()/exit()` when there are multiple threads
- **User thread**: Thread is implemented as a user library (just library calls); kernel is not aware of the threads
  - Implemented by library: more flexible, e.g. customized thread scheduling policy - One thread blocked -¿ all threads blocked - Cannot exploit multiple CPUs
- **Kernel thread**: Thread is implemented in the OS (using syscalls); thread-level scheduling is possible - Multiple threads from same process can run simultaneously - Thread operations are syscalls: more resource-intensive and slower - Less flexible, so it can be generic enough for all multithreaded programs
- **Hybrid thread model**: User thread can bind to a kernel thread



```
int pthread_create(pthread_t *thread,
const pthread_attr_t *attr  /*NULL*/,
void *(*start_routine) (void *)  /*function ptr*/,
void *arg  /*argument for start_routine*/);
int pthread_exit(void *retval);
int pthread_join(pthread_t thread, void **retval);
```

Inter-Process Communication

Shared memory

- **Advantages**:
  Efficient (only the initial steps involves OS)
  Easy to use (shared mem region behaves like normal mem)
- **Disadvantages**:
  Synchronization (of access)
  Implementation is usually harder

```
int shmget(key_t key /*can be IPC_PRIVATE*/,
size_t size, int shmflg /* IPC_CREAT | 600*/);
//IPC_CREAT means memory will be created if nonexistent
void *shmat(int shmid, const void *shmaddr /*NULL*/,
int shmflg /*0*/);
int shmdt(const void *shmaddr);
int shmctl(int shmid, int cmd /*IPC_RMID*/,
struct shmid_ds *buf /*unused for IPC_RMID*/);
```

Message passing

Messages stored in kernel memory space

- **Direct communication**:
  Sender/receiver explicitly names the other party
  One buffer per pair of (sender, receiver)
  Indirect communication:
  Sender sends to mailbox/port
  Receiver receives from mailbox/port
- **Blocking primitives (synchronous)**:
  Send() blocks until message is received
  Receive() blocks until message has arrived
  Non-blocking primitives (asynchronous):
  Send() does not block
  Receive() returns some indication if no message is available
- **Advantages**:
  Portable (can implement in distributed system or network)
  Easier synchronization (blocking primitives implicitly synchronize sender/receiver)
  Disadvantages:
  Inefficient (needs OS intervention)
  Harder to use (messages limited in size/format)

Unix Pipes

Pipes function as fixed-size circular byte buffer with implicit synchronization
- writers wait when buffer is full
- readers wait when buffer is empty

```
int pipe(int pipefd[2]); // create new pipe
pipefd[0]: file descriptor for reading
pipefd[1]: file descriptor for writing
```

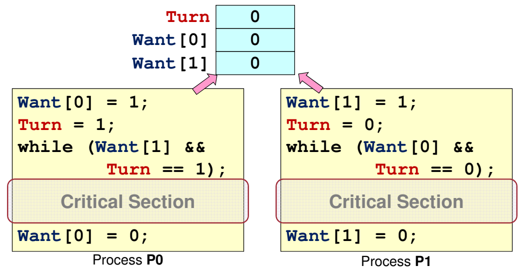
Unix Signals

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
// returns previous signal handler, or SIG_ERR on error
```

Synchronization

Critical Sections

- **Properties of correct implementation**:
  Mutual exclusion: If there is a process in CS then all other process cannot enter CS
  Progress: If no process is in CS then one waiting process should be granted access
  Bounded wait: After a process requests to enter the critical section, there exists an upper bound of number of times other processes can enter the CS before this process
  Independence: Process not in CS should never block other processes
- **Symptoms of incorrect synchronization**:
  Deadlock: All processes blocked
  Livelock: Processes are not blocked, but they keep changing state to avoid deadlock and make no other progress
  Starvation: Some processes are blocked forever
- **Test and Set**: `TestAndSet Reg, Mem`
  - Atomically load current content at `Mem` into `Reg`, and stores 1 into `Mem`
- **Peterson's Algorithm**:



```
Want[0] = 1;
Turn = 1;
while (Want[1] &&
        Turn == 1);
Critical Section
Want[0] = 0;
```

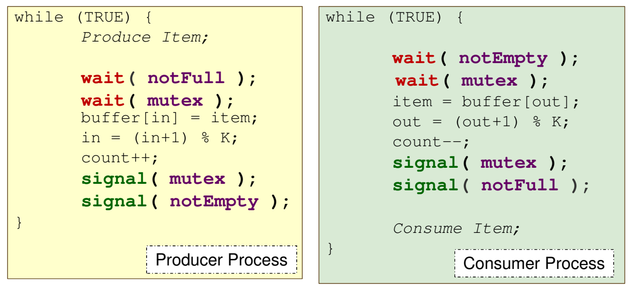
```
Want[1] = 1;
Turn = 0;
while (Want[0] &&
        Turn == 0);
Critical Section
Want[1] = 0;
```

  - Assumption:
    - Writing to `Turn` is an **atomic** operation

  Disadvantages: busy waiting, low level, not general

Semaphores

- **Properties of correct implementation**:
  Mutual exclusion: If there is a process in CS then all other process cannot enter CS
- **Wait()/P()/Down()** and **Signal()/V()/Up()**
- Threads queue up on a semaphore (fair scheduling)
- **General semaphore**: value can be any non-negative integer
- **Binary semaphore**: value can be only 0 or 1 (undefined behaviour to Signal() on binary semaphore which is currently 1)
- **Producer-Consumer**:



```
while (TRUE) {
        Produce Item;

        wait( notFull );
        wait( mutex );
        buffer[in] = item;
        in = (in+1) % K;
        count++;
        signal( mutex );
        signal( notEmpty );
}
```
Producer Process

```
while (TRUE) {

        wait( notEmpty );
        wait( mutex );
        item = buffer[out];
        out = (out+1) % K;
        count--;
        signal( mutex );
        signal( notFull );

        Consume Item;
}
```
Consumer Process

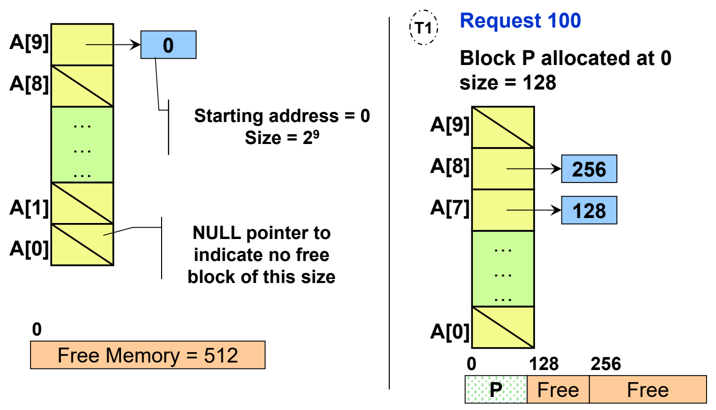
  - Initial Values:
    - `count = in = out = 0`
    - `mutex = S(1), notFull = S(K), notEmpty = S(0)`

Memory Management

- **Memory regions**:
  Text: for instructions
  Data: for global variables
  Heap: for dynamic allocations
  Stack: for function invocations
- **Transient data**: variables with automatic storage duration
  Persistent data: globals, dynamically allocated memory
- **Alternatives for memory abstraction**:
  Address relocation: translate all addresses at load time
  Base + Limit registers: generate instruction to add *Base* to all memory references at compile time, and check against *Limit* for validity
- **Memory partitioning**: every process gets a contiguous memory region
  Fixed partitioning: physical memory is split into fixed number of partitions, each process occupies exactly one partition
  - *Internal fragmentation* when process does not need whole partition
  Dynamic partitioning: partition is created based on actual size of process, OS keeps track of memory regions, and split/merge free regions when necessary
  - *External fragmentation* unused "holes" in physical memory due to process creation/termination

Dynamic Allocation Algorithms

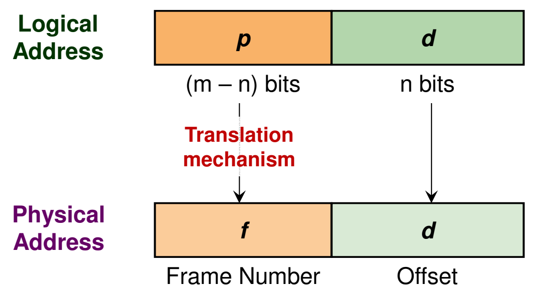
- **Linear search based**:
  First-Fit: take the first hole that is large enough
  Best-Fit: take the smallest hole that is large enough
  Worst-Fit: take the largest hole
  Merging & Compaction:
    When partition is freed, try merging with adjacent holes
    Can move occupied partitions around to consolidate holes
- **Buddy system**:



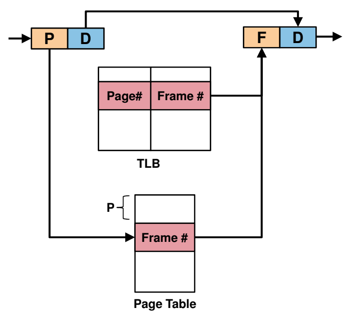
  - Each element $A[J]$ is a linked list that keeps track of free blocks of size 2^J
  - Each free block is indicated just by the starting address
  - There might be a smallest allocatable block size, i.e. a constant $K > 0$ such that $A[J]$ exists only when $J \geq K$
  - To allocate size 2^S: find smallest free block with $size \geq 2^S$, then repeatedly split until there is a block of size 2^S, then return that block of size 2^S
  - To deallocate: if buddy is also free, merge with buddy and repeat; otherwise add block to the linked list

Paging

- **(Physical) frame**
  (Logical) page
  - Frames and pages have the same size
  - Logical memory remains contiguous but physical memory may be disjoint
- **Address translation**:
  - Make frame size (= page size) a power-of-2



- **Fragmentation**: Paging removes external fragmentation (all free frames can be used without wastage), but pages can still have internal fragmentation (logical memory required may not be a multiple of page size)
- **Page table**:
  Stores physical frame for each logical page
- **Translation look-aside buffer (TLB)**:
  cache of *a few* table entries



- **Memory access time with TLB**:
  $= TLBhit + TLBmiss$
  $= 40\% \times (1\text{ns} + 50\text{ns}) + 60\% \times (1\text{ns} + 50\text{ns} + 50\text{ns})$
- **Context switching & TLB**: On context switch:
  - TLB entries are flushed (so incoming process won't get incorrect translation)
  - (Optional) Save TLB state to PCB, and restore TLB data for incoming process (to reduce TLB misses)
- (x86) On a TLB miss, the hardware searches through the page table (without invoking the OS); OS is informed only on page fault
- **Extensions for protection**:
  Access-right (RWX) bits: memory access is checked against access right bits (by hardware)
  Valid bit: represent invalid logical addresses, invalid access will be caught by OS
- **Page sharing**: Several processes use same physical frame
  - e.g. shared libraries, copy-on-write from fork()

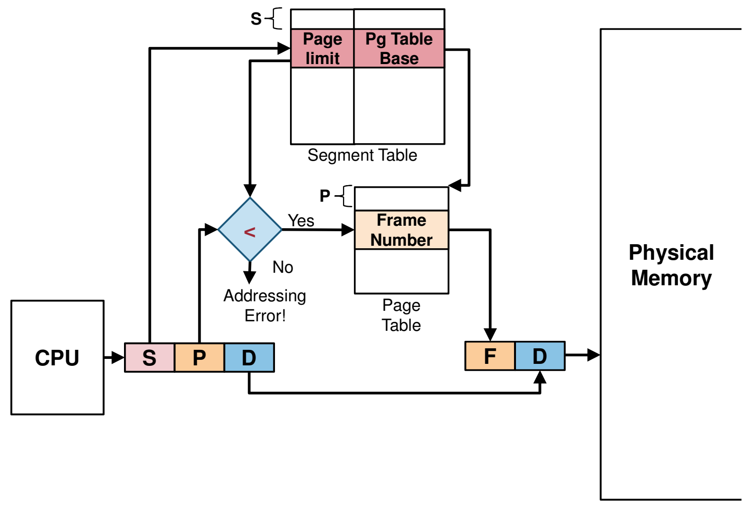
Segmentation

- Each region of memory is placed in a separate segment so they can grow/shrink freely
- Each memory segment has a *segment id* and *limit*
- Memory references are specified as:
    segment id + offset
- Can cause external fragmentation

	Base	Limit
0	3500	2200
1	6000	1500
2	2400	1100
3	0	1300

segment table

Segmentation with Paging



Secondary Storage *(With Paging)*

- Some pages can be stored on secondary storage, so that a process can use more logical memory than what is physically available
- **Page table** stores *memory resident* bit:
  - memory resident: page in physical memory (RAM)
  - non-memory resident: page in secondary storage
- **Page fault**: When CPU tries to access non-memory resident page
  - OS locates the page in secondary storage and loads it into physical memory
- **Thrashing**: Page fault happens too often
  - for well-behaved programs it is unlikely to happen due to temporal and spatial locality

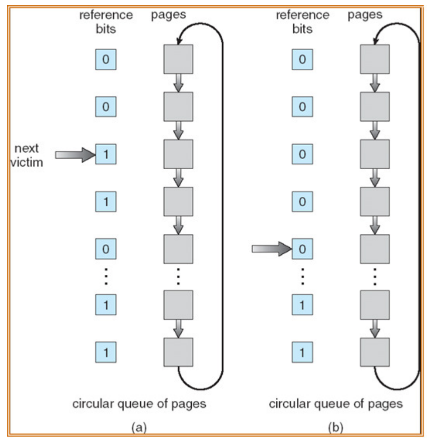
Page Table Structure

- **Direct paging**: All pages in single table, might occupy several memory pages
- **2-level paging**: Keep a page directory, [[TODO]]
- **Inverted page table**: Single table for all processes, stores (*pid*, *logical page*) indexed by frame number

Page Replacement Algorithms

- **Optimum (OPT)**: Replace the page that will not be used again for the longest period of time, not feasible as it needs future knowledge
- **First In First Out (FIFO)**: Evict the oldest page first
  - simple to implement, OS maintains a queue of resident page numbers
  - can be tricked: try 3 / 4 frames with 1 2 3 4 1 2 5 1 2 3 4 5 (Belady's Anomaly)
  - does not exploit temporal locality
- **Least Recently Used (LRU)**: Evict the page that has not been used for the longest time
  - makes use of temporal locality
  - does not suffer from Belady's Anomaly
  - difficult to implement, needs hardware support:
  (option 1) store "time-of-use" and update it on every access, need to search through all pages to find earliest time-of-use
  (option 2) maintain a "stack"; when page is accessed, remove from stack (if exists) and push on top of stack

- **Second Chance Page Replacement (CLOCK)**: Maintain a circular queue of page numbers, and each page table entry has a "reference bit"
  - General Idea:
    - Modified FIFO to give a second chance to pages that are accessed
    - Each PTE now maintains a "reference bit":
      - 1 = Accessed, 0 = Not accessed
    - Algorithm:
      1. The oldest FIFO page is selected
      2. If reference bit == 0 ➔ Page is replaced
      3. If reference bit == 1 ➔ Page is given a 2nd chance
         - Reference bit cleared to 0
         - Arrival time reset ➔ page taken as newly loaded
         - Next FIFO page is selected, go to Step 2
    - Degenerate into FIFO algorithm
      - When all pages has reference bit == 1



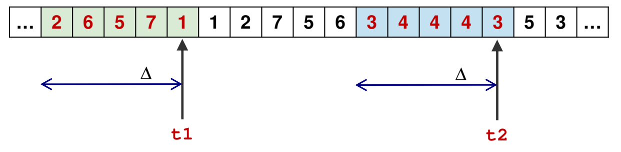
- Use **circular queue** to maintain the pages:
  - With a pointer pointing to the oldest page (the **victim page**)
- To find a page to be replaced:
  - Advance until a page with '0' reference bit
  - Clear the reference bit as pointer passes through

Frame Allocation

- **Simple Approaches**:
  Equal allocation: Each process gets same number of frames
  Proportional allocation: Each process gets number of frames proportional to its memory usage

- **Local/Global Replacement**:
  Local replacement: Evicted page selected from same process
  - thrashing limited to single process
  Global replacement: Evicted page can be from any process
  - can cause thrashing in other processes

- **Working Set Model**:
  - Set of pages referenced by a process is relatively constant in a period of time *("locality")*
  - Page fault only when changing to new locality
  - Use magic constant $\triangle$ = working set window (interval)



- Assume
  - Δ = an interval of 5 memory references
- **W(*t1*,Δ)={1,2,5,6,7}** (5 frames needed)
- **W(*t2*,Δ)={3,4}** (2 frames needed)

File System Management

- **Access types**: Read, Write, Execute, Append, Delete, List (retrieve metadata of the file)

- **Access control list (ACL)**: list of user identities and allowed access types (very customizable but use large space)

- **Permission bits**: Owner/Group/Universe, Read/Write/Execute, e.g. `rwxr--r--`

- In Unix, Minimal ACL = permission bits, Extended ACL = add named users/groups

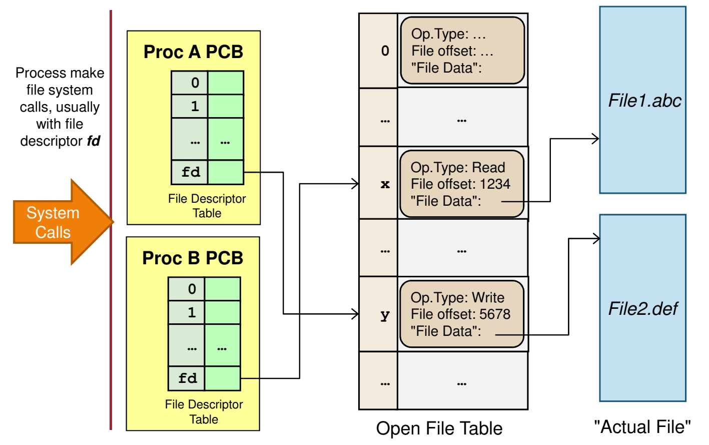
- **File structure**:
  Array of bytes (usual)
  Arr. of fixed-length records (can jump to any record easily)
  Arr. of var.-length records (flexible but hard to find record)

- **Access methods**:
  Sequential access: have to read and rewind on order
  Random access: read in any order, exposed via either way:
  1. `Read(Offset)`: every read explicitly states position
  2. `Seek(Offset)`: special operation to move to new location
  Direct access: like random access, but for fixed-length records (e.g. in database)

- **Generic operations on file data**: Create, Open, Read, Write, Reposition/Seek, Truncate

- **Info kept for opened file**: File pointer (current location in file), Disk location, Open count (number of processed that has this file opened)

- **Open file table (Unix)**:

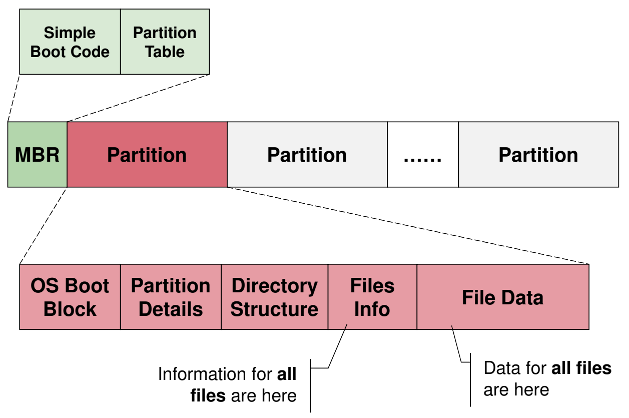


- Two file descriptors (i.e. open file table entries) can point to same file (e.g. two process open same file, or same process open file twice)
- Two processes use the same file descriptor (i.e. open file table entry) (e.g. after `fork()`)

- **Links in directory structure)**:
  Hard link (limited to file only): ref counted, creates DAG
  Symbolic link (can be file or directory): uses special link file, can create general graph

File System Implementations

- **Generic disk organization**:



Block Allocation

- **Contiguous**: allocate consecutive disk blocks to a file
  - simple to keep track, fast access (no need to seek)
  - has external fragmentation

- **Linked list**: each disk block stores next block number too
  - no external fragmentation
  - slow random access to file, part of block is used for pointer

- **File allocation table (FAT)**: next block numbers stored in single table that is always in memory
  - faster random access
  - FAT keeps track of all disk blocks (takes up memory space)

- **Indexed allocation**: each file has an index block (stores list of blocks containing the file)
  - lesser memory overhead, fast direct access
  - limited max file size, index block overhead

- **Indexed allocation with linked list**: index block contains a pointer to next index block (of the same file)
  - no file size limit

- **Multi-level index**: like multi-level paging
  - very large file size limit
  Combination: e.g. Unix I-node

Free Space Management

- **Bitmap**: Each disk block represented by 1 bit
  - e.g. 1=free, 0=occupied

- **Linked list**: Use an unrolled linked list of disk blocks
  - store the free list in free disk blocks

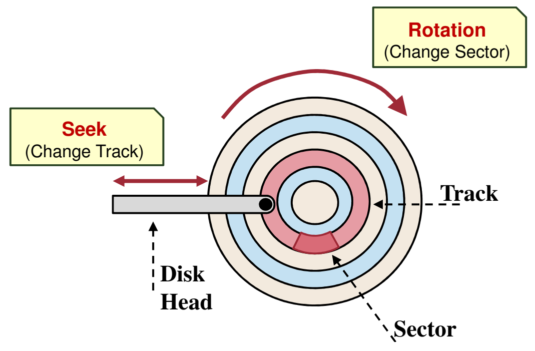
Directory Implementation

- sub-directory is usually stored as file entry with special type in a directory

- **Linear list**:
  - requires linear search, usually last few searches are cached

- **Hash table**:
  - file name is hashed
  - fast lookup, but hash table has limited size and depends on good hash function

Disk Scheduling



- **First-Come-First-Serve (FCFS)**

- **Shortest Seek First (SSF)**: closest track first

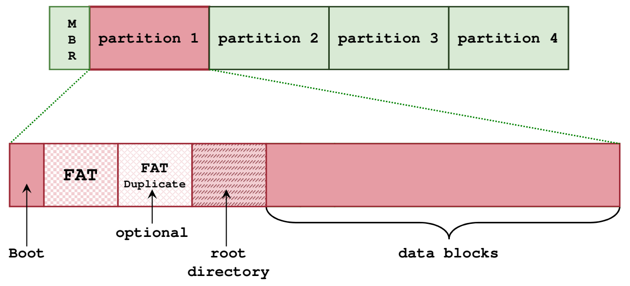
- **SCAN** (elevator), **C-SCAN** (outside to inside only)

- **LOOK** (real elevator)

File System Case Studies

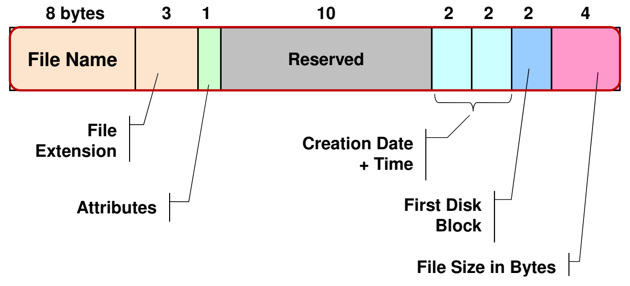
FAT

- **Layout**:



- **File allocation table** contains one of:
  - **FREE**
  - <Block number> of next block
  - **EOF**
  - **BAD**

- **Directory entry**:
  - special type of file
  - root directory is stored in a special location, other directories stored in normal data blocks



- **File deletion**: set first letter of filename to 0xE5

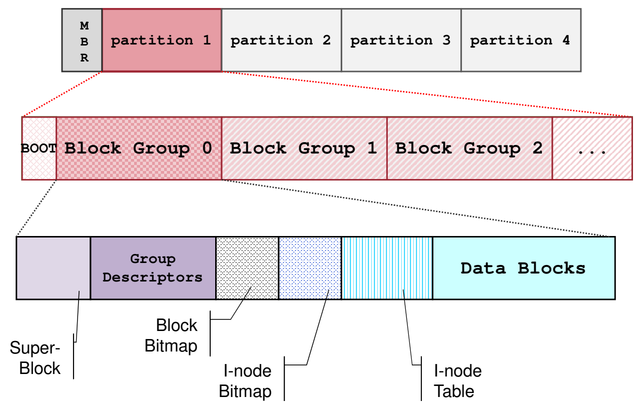
- **Free space**: must be calculated by going through FAT

- **Clusters**: (for newer FATs) group of disk blocks as smallest allocation unit

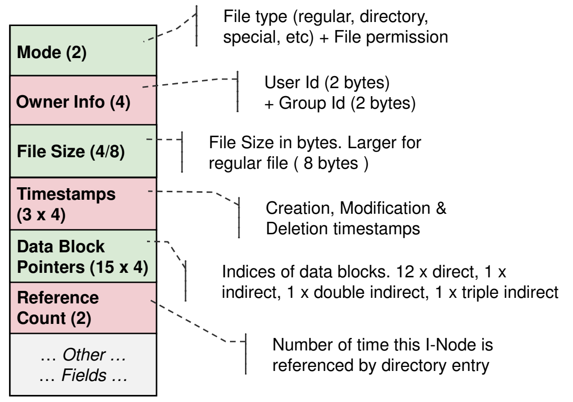
- **Virtual FAT**: use multiple dir. entries for long file name

Ext2

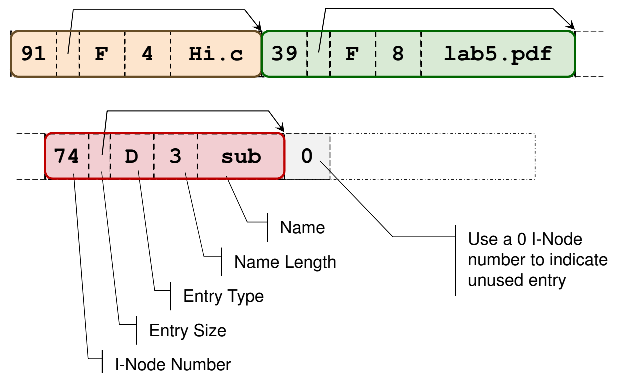
- **Layout**:



- **Superblock, group desc.** duplicated in each block group
- **Block, I-node bitmap** 1=occupied, 0=free
- **I-Node structure**:


Mode (2)
Owner Info (4)
File Size (4/8)
Timestamps (3 x 4)
Data Block Pointers (15 x 4)
Reference Count (2)
… Other … … Fields …
File type (regular, directory, special, etc) + File permission
User Id (2 bytes) + Group Id (2 bytes)
File Size in bytes. Larger for regular file (8 bytes)
Creation, Modification & Deletion timestamps
Indices of data blocks. 12 x direct, 1 x indirect, 1 x double indirect, 1 x triple indirect
Number of time this I-Node is referenced by directory entry

- **Directory entry**:
  - size includes all subfields and possible gap to next entry
  - root directory has a fixed I-node number



- **Deleting a file**:
  - remove its directory entry from the parent directory by adjusting previous size to point to next entry
  - update I-node bitmap by marking file's I-node as free
  - update block bitmap by marking file's blocks as free
- **Hard link**: multiple directory entries point to same I-node
- **Sym. link**: file (not I-node) content is path of target file
  - can become invalid if target is deleted

Journaling

Write information or actual data to separate log file before performing file operation, so it can recover from system crash