CS2106 2324 Sem 2 Final Cheatsheet, by Stanve Widjaja. (Main ref: Hanming)

1 Intro to OS

Simple Defn. Program that acts as intermediary between computer user (programs) and computer hardware. OS evolves with computer hardware and user application/usage pattern. First computers no OS; reprogram by changing physical config of hardware. Advantage of no OS: minimal overhead, but disadvantage of no OS: not portable and inefficient use of computer.

Motivations for OS. Large variation in hardware configs. Hardware in same category has well defined and common functionality (e.g. hard disk, always used to store and retrieve information).

OS serves as an abstraction: hide low level details, present common high level functionality to user. So user can perform essential tasks through OS, no need to concern with low level details. Provides efficiency and portability. OS also serves as resource allocator: prog exec requires mult resources: CPU, memory, I/O devices, etc. For better utilization of resources, multiple programs should be allowed simultaneous execution (manage all of them and arbitrate potentially conflicting requests, for efficient and fair resource use). OS serves as control program: misuse computer due to bugs or malware (prevent errors and improper use of computer, provides security).

OS structures. We have identified major capabilities of OS (specs of OS). Organization important factors: flexibility, robustness, maintainability. OS is essentially a software (runs in kernel mode, possess complete access to all hardware resources), while other software executes in user mode (limited or controlled access).

Generic OS components. Hardware ↔ OS: OS executing machine instr. Hardware ↔ library/user prog: normal machine instr executed (lib/prog code). OS ↔ lib/user prog: calling OS using syscall interface. User prog ↔ library: user prog calls library code. System processes: provide high level services, usually part of OS.

OS as program. Known as kernel: program with special features (hardware issues, syscall interface, code for interrupt handlers/devide drivers). Has to be different than normal programs (can't use syscall in kernel code, normal libraries or normal I/O).

OS Structures. Monolithic and Microkernel, cover these two in details as most other approaches are variant or improvement. Monolithic: kernel is BIG special program, good SE principles still possible with modularization and separation of interfaces/imple. Adv: well understood, good performance, disadv: highly coupled components, devolved into (very) complicated internal structure. Microkernel: small and clean, provides basic and essential facilities (IPC, address space or thread management), higher level services (runs as server process outside of the OS, built on top of basic facilities). Adv: kernel usually more robust and extendible, better isolation. Disadv: lower performance.

Virtual Machines/VMs. OS hard to debug, so we test potentially destructive imple by VM (software-emulation/virtualization of (underlying) hardware). Also known as Hypervisor.

Type 1: provides individual VMs to guest OSes. Type 2: runs in host OS, guest OSes runs inside VM.

2 Processes

Abstraction for running program, identified by PID.

Efficient Hardware Utilization. OS should provide efficient use of hardware resource. Allow multiple programs to share hardware (multiprogramming, time-sharing). Why Process? OS should be able to switch from running program A to program B, and it requires "info regarding execution of program A needs to be stored (somewhere)", and "program A's info replaced with info needed to run program B".

Process Abstraction. Dynamic abstraction for program execution. Memory context: {Code, Data}, Hardware context: {Registers, PC}, OS context: {Process Properties, Resources Used}. [RECAP]: Memory "storage for instruction and data", cache "duplicate part of memory for faster access, usually split into instruction cache and data cache", fetch unit "loads instruction from memory, indicated by special register (PC/program counter)", functional units "carry out instruction execution, dedicated to diff instruction type", registers "internal storage for fastest access speed {GPR (general purpose register) is "accessible by user program (visible by compiler), Special Register (e.g. Program Counter)}.

Function Call. Control Flow and Data: if f() calls g(), then f() is caller and g() is callee. Important steps: setup parameters, transfer control to callee, setup local variable, store result if applicable, return to caller.

Introducing stack memory: the region is the new memory region to store info regarding function invocation, described by a stack frame. It contains return address of caller, arguments/params for function, storage for local vars, other info (stack ptr: first unused location is logically indicated by SP; stack frame added on top when fn is invoked "stack grows", stack frame removed from top when fn call ends "stack shrinks", frame ptr: facilitate access of various stack frame items; stack ptr hard to use as it can change; some processors provide dedicated register for FP; points to fixed location in stack frame; other items accessed as displacement from FP).

There are different ways to setup stack frame: known as fn call convention (info stored in stack frame or registers? portion of frame prepared by caller/callee? portion of frame cleared by caller/callee, caller or callee adjust SP? No universal way, hardware and prog-lang dependent).

Dynamically Allocated Memory. Most PL allow dynamically allocated mem, i.e. acquire memory space during execution time. Example: C uses malloc() fn call, C++ and Java uses "new" keyword.

We can't use data/stack mem regions since they have different behaviors (allocated at runtime, size not known at compile time → can't place at data region; no definite deallocation timing "can be explicitly freed in C/C++, implicitly freed by garbage collector in Java" → cannot place in stack region). Heap memory lot tricker to manage due to its nature (variable size, variable allocation/deallocation timing; can construct scenario where heap mem allocated in such a way to create "holes" in memory).

P-State. 5-stage proc model: New, Ready, Running, Blocked, Terminated. Transitions: Create (NIL to New), Admit (New to Ready), Switch (Ready to Running), Switch (Running to Ready), Event Wait (Running to Blocked), Event Occurs (Blocked to Ready).

P-Table. Each process has PCB (proc-ctrl-block) that stores entire execution context (for that process. These blocks stored in Process Table by a kernel.

OS interaction with Process. C/C++ prog can call library version of syscalls. Function wrapper has same name and params, function adapter is user-friendlier.

Syscall mechanism. Prog invokes syscall, libcall places syscall num in register, libcall executes TRAP switch kernel mode, syscall handler determined using number by dispatcher, syscall handler executed and completed, OS switches to user mode and returns ctrl to libcall, libcall returns to user prog using fn return.

Exceptions, Interrupts. The former are synchronous, causes exception handler to execute. Examples of exceptions (in each stage of MIPS? execution) include {IF: memory fault, RF: illegal instruction, ALU: arithmetic exception, MEM: memory fault, WB (writeback): (none???)}. The latter are async, suspends program, executes interrupt handler.

Handler Routine. Save register/CPU state, perform handler routine, restore register/CPU state, return from interrupt.

UNIX context. UNIX proc abstraction has info on PID, state, parent PID, cumulative CPU time, etc.

UNIX operations. init: root of all procs in UNIX, created in boot-up, has PID 1. fork(): creates child proc with copy of parent's executable image, data not shared, fn returns 0 for child, and > 0 for parent. *exec(): replaces current executing process with new one; only replaces code; PID and other info remains. execl(char path, char arg0, arg1, ..., argN, NULL): path is location of executable, argX command line arguments, NULL is end of argument list flag. execv(char path, char argv) is same as execl() but arguments are passed through an array instead of individually. exit(): takes in status, terminates process, returns status to parent process; most programs have implicit exit(); upon process termination, some resources are not released (PID, status, CPU time, etc; generally PCB remains). wait(): waits for a child (and blocks), if a child exists, else returns -1. Cleans up on child process, e.g. removes PCB of child.

Zombie: child finishes but parent not call wait(). Orphan: subset of zombie, when parent terminates before child process; init becomes the parent and does the clean-up.

3 Proc-Scheduling

Concurrent execution ensures multiple processes progress in execution; can be pseudoparallelism (virtual/illusory parallelism) or physical parallelism (multiple CPU/multi-core CPUs). Can assume the 2 forms are not distinguished (in this lecture).

Processing Environment. Batch Proc, Interactive (multiprogramming), real time proc (have deadlines to meet, usually periodic process).

Common Criteria. Fairness: gets fair share of CPU time per proc/per user; no starvation. Balance: all parts of computing system should be utilised.

Types of sched algo. Non-preemptive/cooperative: process runs until blocks or gives up CPU voluntarily. Preemptive: proc given a fixed amount of time, possible to block or give up early.

Step-by-step scheduling. Scheduler triggered (OS takes over), if need context switch: context of curr run proc is saved and placed on blocked/ready queue, pick suitable process P, setup context for P, let P run.

Illustration for Scheduling. Each proc has diff requirement of CPU time (its behavior), there are many ways to allocate (influenced by process environment), AND there are a number of criteria to evaluate the scheduler. The behaviors are CPU-activity: computation/number-crunching (compute-bound proc), IO-activity: request/receive service from I/O devices, e.g. print to screen, read from file, etc (IO-bound proc).

Batch Processing Algorithms. No user interactions, non-preemptive scheduling is predominant.

Batch Procesing Criteria. Turnaround time: finish time - arrival time, throughput: num tasks finished per unit time ("num of tasks in a particular set/time needed to execute all tasks in that set"), CPU utilisation: percentage of time when CPU working on a task. [BONUS] includes makespan: time taken to complete ALL jobs.

FCFS: FIFO queue based on arrival time. Properties: no starvation (task in front of task X always decreasing, assuming all tasks complete), convoy effect (one long task at start can GREATLY increase turnaround time for short tasks (waiting) behind).

SJF (shortest job first): select task with smallest total CPU time. Properties: minimizes avg wait time (Proof: MATH), possible starvation (long jobs may not get a chance), difficult to predict [the actual time needed] (Need to predict; can be calculated using $\text{Predicted}_{n+1} = \alpha\text{Actual}_n + (1-\alpha)\text{Predicted}_n$.

SRT (shortest remaining time): self-explanatory. Preemptive [version; I think not all SRT are preemptive]: new short job can preempt long running job.

Interactive Algorithms: preemptive scheduling done to ensure good response times. Thus, scheduler needs to run periodically.

Interactive Criteria. Response time: (first?) time system (CPU?) responds minus request time. Predictability: less variation means more predictable.

Question to answer: how scheduler take over CPU periodically (ITI/interval for timer interrupt; timer interrupt handler invokes scheduler), how to ensure user prog can never stop scheduler from executing (time quantum: execution duration given to proc, can be constant/variable among processes; must be multiple of ITI, large range of values, commonly 5ms to 100ms)?

RR (round robin): dequeue and enqueue from queue, giving each proc a fixed TQ. Blocked tasks queued elsewhere until ready, then it will be queued back. Response time guarantee: n-th task run by $q(n-1)$. Choice of TQ: if big, better CPU utilisation but longer wait time, if small, more overhead from context switch but shorter wait time.

Priority Scheduling: assign task a priority value, select task highest priority. If preemptive, higher priority task can preempt running ones with lower priority. If not preemptive, it will wait for next round of scheduling. Handles priority: some tasks more important. Starvation: low priority proc can starve, especially if preemptive; 2 ways to fix: derease priority of curr run proc every TQ, stop run proc after TQ and exclude it from next round. Generally, hard to guarantee or control exact amount of CPU time given to proc using this algo.

Priority inversion: lower priority task preempts a higher priority one. {A = 1, B = 3, C = 5}, 1 is highest. C starts and locks resource (e.g. file), B preempts C, C

unable to unlock file, A arrives and need same resource as C (but resource is locked!), B continues executing even if A has higher priority.

MLFQ: designed to solve 1 BIG + HARD issue (schedule without perfect knowledge; most algo requires certain info: proc behavior, runtime, etc). MLFQ is adaptive: "learn proc behavior automatically" and minimizes both "resp time for I/O-bound proc, turnaround time for CPU-bound proc".

Rules: [BASIC RULES] if priority(A) > priority(B), A runs. If priority(A) == priority(B), A and B runs in RR. [PRIORITY SETTING/CHANGING RULES] new job → highest priority, if a job fully utilizes time slice → priority reduced, if a job give up/blocks before finishes time slice → priority retained.

Lottery Scheduling: gives out "lottery tickets" to processes, and a random one is chosen when scheduling is done. In long run, process holding X% of tickets will win X% of lottery held, and use resource X% of the time. Responsive: new proc can participate in next lottery. Level of ctrl: such as {a proc can distribute tickets to child proc, an impt proc can be given more tickets, can give different types of tickets: I/O-device or CPU tickets, depending on usage of each resource per task}. Simple to imple: YEAH LOL SELF-EXPLANATORY.

4 Threads

Motivation: process creation using fork() and context switch are expensive. Indep mem space means harder communication as well. We thus want to add threads so multiple parts of same prog are "executing concurrently" (i.e. it is invented to overcome problems with proc model; started out as quick hack and eventually matured to be very popular mechanism).

Basic Idea: one traditional proc has single thread of ctrl; one instr of whole prog executing at any time. We "simply" add more threads of ctrl to same proc (multiple parts of prog executing at same time, conceptually). Illustration: with fork(), we effectively have two "threads" of control (after the fork()).

New model/approach (multi-threading): 1 process can have multiple threads ("multithreading"); threads in same proc shares {memory context {text, data, heap}, OS context {PID, other resources e.g. files}}. They each need unique info: {Identification (usually thread ID), registers (GPR + special), "stack"}.

Context Switch. We alr know proc context switch involves {OS + hardware + memory context}. Thread switch within same proc involves {hardware context (registers + "stack" (actly just changing FP + SP registers))}. Much lighter than process (a.k.a. lightweight process).

Benefits. Economy (much less resources needed), resource sharing (threads share most resources, no need (additonal mechanism) to pass info around), responsiveness (multithreaded progs can appear much more responsive), scalability (multithreaded progs can make use of multiple CPUs). Problems. System call concurrency: parallel system calls are possible → need to ensure correctness. Process behavior: some parts are questionable: {when fork() called, only 1 thread in new process is created (Linux), if single thread calls/executes exit(), all threads exit (Linux), if single thread calls exec(), all threads exit and new executable runs}. These are for Linux, no one knows how to handle these cases in general (discussed in lecture).

Two Major Thread Implementations. User Threads: imple-d as user library (runtime system in process will handle thread-relaed operations). As a result, kernel not aware of all the threads in the process. Adv: {Works on any OS: all programs can have multiple threads, just library calls: threads operations are just libcalls, configurable and flexible: e.g. can have customised thread schedule-ing policy}. Disadv: {OS is not aware: scheduling performed at proc-level, i.e. if 1 thread blocks, proc gets blocked, and all threads block; also cannot exploit multiple CPUs}. Kernel Threads: imple-d in OS, thread operations are handled as syscalls. Thread-level scheduling is possible; kernel will schedule the threads, instead of process. In fact, kernel may make use of threads for its own execution. Adv: {Kernel can schedule on thread levels (multiple threads can run simultaneously on multiple CPUs)}. Disadv: {Thread operations are now syscalls! (slow, expensive, more resource-intensive), [generally] less flexible (used by all multi-threading programs: if too many features, overkill and expensive for simple progs, BUT if too few features, it won't be flexible enough for some programs)}.

Hybrid Models. We in fact can have both models. OS schedule on kernel threads only; and user threads can bind to a kernel thread. This offers great flexibility (can limit the concurrency of any process/user). SIMULTANEOUS multi-threading: when hardware supports parallel execution of threads on same core, by providing multiple sets of registers. Example would be "hyperthreading on Intel processor".

UNIX commands (POSIX Threads — a popular thread API). POSIX thread API defined by IEEE, supported on most UNIX variants. As imple is not specified, *pthread* can be imple-d as user/kernel thread. Header: #include <pthread.h>, compilation gcc XXXX.c -lpthread (flag is system dependent), pthread_t is data type for a pthread id, pthread_attr is data type for thread attributes.

pthread_create() takes in 4 params returns int, 0 for success. The params are {1. Ptr to pthread id, which it will update, 2. ptr to pthr attr to ctrl the behavior of thrd, 3. ptr to fn (start routine) to be executed by thrd; function shld "take in" a void pointer for args, and return void ptr as well, 4. Args for startRoutine fn, as void ptr}.

pthread_exit() takes in void ptr exit value and terminates [i.e. if a "return XYZ;ßtatement is used, then "XYZïs captured as exit val]. If not specified, return value of startRoutine fn is captured as exit val.

pthread_join(): simple synchronization. To wait for termination of another pthread. Takes in pthread id and void ptr ptr (yes it's a pointer to a void/"anything" pointer). Returns 0 if success (and updates the status); which is exit valu returned by target pthread. Note: Thread A calls join(ID of Thread B) ⇒ A wait for B; can think of it as "putting thread B above thread A, with B's return value attached to it".

5 Inter-Process Communication (IPC)

Motivation: processes need to share info, but memory spaces are independent (i.e. it's hard for cooperating processes to share info). So need IPC mechanisms. 2 common IPC mechanisms: shared-memory and message-passing; 2 Unix-specific IPC mechanisms: pipe and signal.

Shr-Mem. P1 (short for Process 1) creates shrd mem region M, P2 attached region to (its) own memory space. They can now write to that region; and data can be seen by both parties (behaves very similar to normal memory region). Also applicable to multiple processes sharing same memory region. OS is involved in creating and attaching M only. ADV: efficient (see previous sentence), ease of use (shr-mem-region behaves like a regular one; can easily write info of any size or type). DISADV: synchro (need to sync the shrd resource), imple usually harder (shag).

Shr-Mem (UNIX). Basic steps of usage: create/locate shrd-mem-regi M, attach M to proc-mem-space, read from/write to M (values written visible to all procs that share M), detach M from mem-space after use, destroy M (only 1 proc need to do this; can only destroy if M not attached to any proc).

Syntaxes: *shmget*() creates shrd-mem-regi, returns (the?) ID; usually done by master program. *shmat*() attaches shrd-mem-regi using ID, and returns ptr to region; done by both master + slave. *shmdt*() detaches shrd-mem-regi using ptr from attaching; done by both master + slave. *shmctl*() destroys shrd-mem-regi using ID; generally done by master prog.

Msg-Passing. P1 preps msg and sends to P2; this msg needs to be stored in kernel-mem-space. P2 receives msg. Both snd, rcv are syscalls (so need to go through OS). Additional Properties: {1. Naming: how to identify the other party in the communication; there are two: {Direct comm: snd-er and rcv-er explicitly name the other party; needs one linkk per pair; processes need to know the identity of other party, Indirect comm: snd and rcv from a port; one port can be shared amonog a num of procs}, 2. Synchronization: {1. Blocking-primitives (synchronous) {*send*(): sender blocked until msg is rcv-ed, *receive*(): rcv-er blocked until msg arrived}, 2. Non-blocking-primitives (asynchronous) {*send*(): sender resume operation immediately, *receive*(): rcv-er gets msg if available, else gets some indication that there is no msg (yet)}}.

ADV: portable (easily imple-d across different processing environments), easy-to-sync (using blocking-primitives force snd-er and rcv-er to be synchronous). DISADV: inefficient (needs OS intervention), hard(er)-to-use (msg usually limited in size and format).

Msg-Passing in UNIX (i.e. UNIX Pipes). "Plumber needed! Leaking pipes all around!" [intro msg in lecture].

Jokes aside, in UNIX, a proc has 3 default comm channels: *stdin* (commonly linked to keyboard inp), *stdout* (commonly linked to screen) and *stderr* (only used to print out error msgs).

The "|" symbol in shell directs one proc's output to another's input (all done "internally"). Created with 2 ends, 1 end for reading, other end for writing. More generally, a pipe is a FIFO circular bounded byte buffer with implicit synchronization - writers wait when buffer is full, readers wait when buffer is empty. Variants: multiple readers + mult writers [note: normal shell pipe has 1 writer, 1 reader], half-duplex/unidirectional [with one write-end + one read-end] vs full-duplex/bidirectional [any end for read + write].

As an IPC mechanism (in UNIX), a pipe can be shared between 2 processes; it is a form of producer-consumer relationship where P produces/writes n bytes, Q consumes/reads m bytes. Behavior is like an anonymous file; and follows FIFO, meaning it must access data in order. If P writes a, then $b,c,d \rightarrow Q$ must read a first, then b,c,d.

UNIX Pipe System Calls. Header file is "#include <unistd.h>" and (declaration) syntax is "int pipe(int fd[])". The declaration takes in integer array of size 2 (e.g. fd[2]; fd stands for file descriptor. We have fd[0] as reading-end, fd[1] as writing-end). It returns 0 to indicate success (and !0 otherwise).

UNIX Pipe syntaxes. *pipe*() alr described. *close*() takes in file descriptor, and closes it; the file descriptor is now unused; 0 is stdin, 1 is stdout, 2 is stderr. *dup*() takes in file-descriptor/fd and finds lowest unused fd, and duplicates it there (i.e. lowest unused fd is NOW an alias for the fd-argument passed into *dup*()). *dup2*() is like *dup*() except you can specify the new file descriptor to duplicate into.

Signal (UNIX). An asynchronous notification (so it is a form of IPC) regarding an event that was sent to a process/a thread. The recipient of the signal MUST handle the (received) signal by "a default set of handlers" OR "a user-supplied handler" (and the latter is only applicable to some signals; i.e. SIGKILL that forces termination CANNOT be handler-replaced). Common signals in UNIX include {Kill, Stop, Continue, Memory error, Arithmetic error, etc}.

We have the syntax of *signal*() as: takes in a signal (number) (that is anticipated) and a handler that [takes in an argument of type *int* and returns *void*], and replaces the default handler by that.

6 Synchronization

Msg-Passing. P1 preps msg and sends to P2; this msg needs to be stored in kernel-mem-space. P2 receives msg. Both snd, rcv are syscalls (so need to go through OS). Additional Properties: {1. Naming: how to identify the other party in the communication; there are two: {Direct comm: snd-er and rcv-er explicitly name the other party; needs one linkk per pair; processes need to know the identity of other party, Indirect comm: snd and rcv from a port; one port can be shared amonog a num of procs}, 2. Synchronization: {1. Blocking-primitives (synchronous) {*send*(): sender blocked until msg is rcv-ed, *receive*(): rcv-er blocked until msg arrived}, 2. Non-blocking-primitives (asynchronous) {*send*(): sender resume operation immediately, *receive*(): rcv-er gets msg if available, else gets some indication that there is no msg (yet)}}.

Synchro-problems. Face them when have concurrent processes execute in an interleaving fashion, and share some modifiable resource. In more details: execution of single sequential proc is deterministic (repeated execution gives same result); execution of concurrent proc may be non-deterministic → we get a "race condition": refer to situations where execution depend on order in which the shr-resources are accessed/modified. SOLUTION is designate segment with race conditions as a CRITICAL SECTION.

Example in Lec-Slides: "Increment 1000 to X" divided into 3: {Load $X \rightarrow$ Reg1, Add 1000 to Reg1, Store Reg1 $\rightarrow X$}.

Critical Section. In a CS, at any point there can only be 1 process executing. Properties of correct CS Imple: {Mutex/mutual-exclusion: if proc P_i executing in a CS, all (other) processes should not be able to enter; Progress: if no proc in a CS, one of the waiting processes should be granted access; Bounded-wait/no-starvation: if proc P asks to enter the CS, there should be an upper-bound on number of times other procs can enter before P does, Independence: process not executing in the CS should not block other procs}.

Symptoms of Incorrect Synchro: {Deadlock: all procs blocked, no progress, it needs ALL the following conditions to be met: {Mutex condition: each resource is either assigned to exactly 1 or 0 proc, Hold-and-wait condition: procs holding resources can ask for more resources, No-preemption condition: cannot forcibly take

away resources previously given, Circular-wait condition: must have circular list of 2 or more procs, each waiting for resource held by the next}, Livelock: in an attempt to avoid deadlock, procs keep changing state and make no other progress (typically process are not blocked); Starvation: some procs are perpetually denied needed resources and cannot make progress (i.e. blocked forever)}.

Implementations. Overview: {Assembly-level-imple-s: mechanisms provided by the processor; HLL/high-level-language imple-s: utilizes only normal programming constructs, High-level-abstraction: provide abstracted mechanisms that provide additional useful features, commonly imple-d by assembly-level-imple-s.}

Assembly-level ("Dw! Processor has all the answers~"). We have assembly-level/machine-level instruction [*TestAndSet*] with "parameters" [*Register,MemoryLocation*]. Behavior: we load current content at *MemoryLocation* into *Register*, AND change the value at *MemoryLocation* to (the immediate/constant) 1. The above is performed as a *single* machine operation (i.e. atomic; cannot be interrupted).

This imple works and meets all 4 criteria. HOWEVER, it empoloys busy waiting (keep checking the condition until it is safe to enter CS) → wasteful use of processing power. Variants of this instr exists on most processors: {Compare-and-Exchange; Atomic-Swap; Load-Link/Store-Conditional}.

HLL-imple ("Using only your brain power :) ..."). Following code segments work on the assumption that writing to *Turn* is an atomic operation. We basically create 2 boolean conditions that prevent a deadlock. Behavior: {Busy waiting (same as above, repeatedly test while-loop condition instead of going to blocked state); low level: error-prone, is not very simple to implement (higher-level prog construct is desirable); hard to generalise: limited to 2 procs (the properties of synchro-mechanism is desirable to generalise — not just mutex)}.

Code for Process i, $i \in \{0,1\}$: $Want[i] = 1$; $Turn = 1$; *while* $(Want[1-i]$ && $Turn == 1)$;

// CRITICAL SECTION

$Want[i] = 0$;

High-level-abstraction ("Let's go meta ..."). SEMAPHORES AND MUTEXES. A generalised synchro mechanism that specify behavior (not implementation). It provides a way to block a number of procs, which will be known as SLEEPING PROCESSES. We can see a semaphore S as a protected integer, with a non-negative initial value. A general semaphore or counting semaphore can have values $S \geq 0$. A binary-semaphore/mutex has values $S \in \{0,1\}$. API: {*wait*(): takes in a semaphore, if semaphore value is ≤ 0, it blocks the current proc and decrements the value, this is an atomic operation; *signal*(): takes in a semaphore, wakes up 1 sleeping proc (if any) and increments semaphore value [regardless if any sleeping process is waken up], this is an atomic operation, and never blocks}.

The invariant is $S_{\text{current}} = S_{\text{initial}} + \#signal(S) - \#wait(S)$, where $S_{\text{initial}} \geq 0$; $\#signal(S)$ is the number of *signal*() executed, $\#wait(S)$ is the number of *wait*() operations completed.

Classical Synchro Problems. Will discuss the problems here. (For solutions, refer to the "Little Book of Semaphores").

PRODUCER-CONSUMER. Procs share bounded buffer of size K. Producers produce items to be inserted in buffer, only when buffer is not full ($< K$ items). Consumers remove items from buffer, only when buffer is not empty (> 0 items). How can we synchronize the two? The following code correctly solve problem, AND, no busy-waiting is used.

Producer code: while(TRUE) {
Produce Item;
wait(*notFull*); *wait*(*mutex*);
buffer[in] = item; in = (in + 1) % K; count++;
signal(*mutex*); *signal*(*notEmpty*); }

Consumer code: while(TRUE) {
wait(*notEmpty*);
wait(*mutex*);
item = buffer[out]; out = (out + 1) % K; count–;
signal(*mutex*); *signal*(*notFull*);
Consume Item;
}

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

READERS WRITERS. Procs share a data structure D, where readers can access and read info from D together, while writers must have exclusive access to D to write info. How can we synchro the two? And how do we prevent the writer from starving; i.e. readers keep reading (latter qn not discussed in lecture). Simple Version of Code:

Writer code: while(TRUE) {
wait(*roomEmpty*);
Modifies data
signal(*roomEmpty*);
}

Reader code: while(TRUE) {
wait(*mutex*);
nReader++; if (nReader == 1) *wait*(*roomEmpty*);
signal(*mutex*);
Reads data
wait(*mutex*);
nReader–; if (nReader == 0) *signal*(*roomEmpty*);
signal(*mutex*);
}

Initial Values: *roomEmpty* = $S(1)$; *mutex* = $S(1)$; nReader = 0;

DINING PHILOSOPHERS. 5 philosophers are seated around a table, and there are 5 single chopsticks placed between each pair of philosophers. When any philo wants to eat, he/she will have to acquire both chopsticks from his/her left AND right. How can we have a deadlock-free and starvation-free way to allow the philo-s to eat freely? (I won't bother writing the code here, it's too long!)

7 Contiguous Memory Allocation

Memory Hardware. Physical mem storage: Random Access Memory (RAM) → can be treated as an array of bytes, and each byte has a unique index (known as PHYSICAL ADDRESS). A contiguous memory region := an interval of consecutive addresses.

The Memory Hierarchy: The right-er it is, the {faster; smaller; more expensive} it is. The things are {Offline storage: {very slow, in seconds; potentially huge (PBs); least expensive}; Hard-Disk: {slow (10 ms); very large (2 TB); very inexpensive ($0.0025/MB)}; RAM: {fast (100 ns); large (8 GB); inexpensive ($0.58/MB)}; Cache: {very fast (10 ns); small (12 MB); very expensive ($150/MB)}; CPU-Registers {very fast (1 ns); very small (512 Bytes); very expensive (part of CPU)}}.

RECAP: mem-usage of proc: {Text (for instr-s); data (for global var-s); heap (for dynamic allocation); stack (for fn variables)}.

The OS needs to perform following tasks: {Allocate mem-space to a new-proc; manage mem-space for proc; protect mem-space between procs; provide mem-related syscalls; manage mem-space for internal use}.

Binding of Memory Address: executable typically contains "code" (for text region), "data layout" (for data region). Summary of "Memory Usage": Generally, 2 types of data in a proc: {Transient-Data: valid only for a limited duration, e.g. during a function call (e.g. parameters, local vars); Persistent-Data: valid for the duration of the program, unless explicitly removed (if applicable), e.g. global var, constant var, dynamically allocated memory}. Both types of data sections can grow/shrink during execution.

No Memory Abstraction. Pros: {straightforward, fast, addresses fixed during (i.e. unchanged after) compile time}. Cons: {Both procs assume they start at 0, resulting in conflicts; it is also hard to protect memory}. Solution 1: relocate addresses: recalculate memory references when procs is loaded into mem, e.g. if proc B is located at addr 8000, add 8000 to all mem addr-s; however, loading time is slow and it's not easy to distinguish a mem-addr from any arbitrary int. Solution 2: base + limit registers: Use a special BASE REGISTER that stores start-addr of proc's mem-space. All mem-references will be offset by this reg-value; We then use a LIMIT REGISTER to indicate the range, i.e. cannot access past the limit; However, there is a lot of overhead since we need to perform an addition and comparison per access; this is later generalised in segmentation.

Physical and Logical Addresses. Generally, embedding physical addr-s in programs is a bad idea. We thus let each proc have a self-contained, independent logical mem-space that they will reference using logical-addr-s, then the OS will do the mapping. We then need some ways to do partitioning of the mem, so that we can switch between proc-s using the different mem-partitions. When the phys-mem is full, we can either remove partitions used by term-ed proc-s or swap the blocked proc to secondary storage. The following algos work on the following assumptions: {each proc uses a contiguous mem-region; the phys-mem is large enough to contain 1 or more proc-s with completed mem-space}.

Fixed-Size Allocation Algos. Phys-mem is split into a fixed num of partitions, then a proc will occupy 1 of them. Behaviors: {Internal fragmentation: when a proc does not occupy the (its) entire partition; easy to manage and fast to allocate: just give any free partition, it's all the same; need to fit largest proc: this also means ALL the smaller proc-s will waste space}.

Variable-Size Allocation Algos. Also known as dynamic partitioning. We allocate just the right amount of space needed, forming HOLES between such partitions. Behavior: {No internal frag: all proc-s get the exact space required; external frag: when holes between proc-s become unusable, waiting space, can be fixed via compaction, but slow (still better than internal, since internal is unreachable/unfixable); need to maintain more info (in the OS); slower to allocate (need to find appropriate region)}.

Dynamic/Var-Size Allocation Algos — the actual algos. Algos: {FIRST-FIT ALLOC: takes the first hole that's large enough, split it into 2, give the process the required space, and the remaining space is a new hole; BEST-FIT ALLOC: take smallest hole that is large enough, rest is same as above; WORST-FIT ALLOC: take largest hole (why? so that have lots of medium-sized holes), rest is same as above; DEALLOCATION AND COMPACTION: try to merge freed partition with adjacent holes (we can also do compaction, i.e. move partitions around, BUT compaction is expensive and should not be done frequently}. Partition Info: partition info can be maintained as either a linked list or a bitmap; for a linked-list, each node contains {1. Status: TRUE = occupied, FALSE = free; 2. Start Address; 3. Length of partition/hole; 4. Pointer to next node}.

BUDDY SYSTEM. Popular imple of multiple free lists. Buddy memory allocation provides efficient {1. Partition splitting; 2. Locating good match for a free partition (hole); 3. Partition de-allocation and coalescing}. Main idea: {Free block is split into half repeatedly to meet reqest size (2 halves form sibling/buddy blocks); when buddy blocks are both free, can be merged to form larger block}. Illustration: [16] split into [4,2,2,8] (actual partitions in memory) but the 2,2 are buddies, and so are the 4,4 and 8,8.

Imple: Keep an array $A[0...K]$ where 2^K is largest block size that can be allocated: {each array ele $A[J]$ is linked list which keep tracks of free block(s) of the size 2^J; each free block is indicated just by the start-addr}; in actual imple, there may be a smallest block size that can be alloc-ed as well: {a block that is too small is not cost effective to manage; will ignore this in the discussion}.

Allocation Algo: to allocate a block of size N: {1. Find smallest s, s.t. $2^s \geq N$; 2. Access $A[S]$ for a free block: {(a) if free block exists, a(i) remove the block from free block list, a(ii) allocate the block; (b) else, b(i) find smallest R from $S+1$ to K s.t. $A[R]$ has a free block B, b(ii) For $(R-1$ to $S)$ do [repeatedly split $B \rightarrow A[S...R-1]$ has a new free block]; Goto Step 2}}.

Deallocation Algo: to free a block B: {1. Check in $A[S]$ where $2^s ==$ size of B; 2. if the buddy C of B exists AND free: {(a) remove B, C from list; (b) merge B and C to get a larger block B'; (c) Goto step 1, where $B \leftarrow B'$}; 3. Else (i.e. buddy of B is not free yet): insert B to the list in $A[S]$}.

Example: Available mem = 512 = 2^9. We have $A[9]$ to $A[0]$, and $A[9]$ "points to" (i.e. have the value) 0. T1. Request 100: block P allocated at 0, size = 128 (get 128 from $2^s = 128, s = 7$); we have $A[9]$ points to 0 (REMEMBER THAT A IS ARRAY OF LINKEDLIST HEADS), $A[8]$ points to 256, $A[7]$ points to 0 points to 128; we now have 0 to 128 is occupied by P, the rest 2 (contiguous) blocks are free. T2. Request 250: block Q alloc-ed at 256, size = 256 ($s = 8$); $A[8]$ now points to 256 (TODO: run through the algo yourself to confirm what's happening), $A[7]$ to 128. Free block is only 128 to 256 now. T3. Free block P (almost self-explanatory); "Hi Buddy!" "Long time no see!"

Where is my Buddy? Observe that {Block addr A is in the format $xxxx00..00_2$; splitting A into 2 blocks of half the size: $B := xxxx00..00_2$ and $C := xxxx10..00_2$} (flip one bit in k position). So, two blocks B and C are buddy of size 2^s, if {lowest S bits (bits $0..S-1$) of B and C identical; AND bit S of B and C is different}.

8 Disjoint Memory Schemes

Now we remove 1 assumption - that proc-mem-space is contiguous. They can now be in disjoint physical mem-loc. This is achieved via paging. [RECAP: second assumption (removed in L9) is that physical memory large enough to contain 1/more proc-s and their entire mem-space.]

OVERVIEW. (this is only in lecture slides) This paragraph is probably the most important one in this section; for each of {PAGING SCHEME, SEGMEN-

TATION SCHEME}, the lecture divides (and conquers) into 4 parts: {P-Scheme: {Basic Idea; Logical-to-Physical Address Translation; Hardware Support; Protection & page sharing}; S-Scheme: {Motivation; Basic Idea; Logical-to-Physical Address Translation; Hardware Support}}.

Paging. The physical mem is split into PHYSICAL FRAMES, and logical mem is split into LOGICAL PAGES of the same size (i.e. size of page == size of frame; so a page can be loaded into any frame). Frame size decided by hardware (mostly by MMU/memory management unit). At execution time, pages of a proc are loaded into any available mem-frame. So, logical mem can remain contiguous, while occupied physical mem-region can be disjoint (each page can be mapped into disjoint frames).

Example, from Lec Slides: Say the instruction is "$loadr1,[2106]$", logical memory of the process (executing the code) consists of 4 pages of size 1000B, and physical memory is 8000B with frame [0,1,2,...,7] containing [E,2,0,E,E,3,E,0]; E:= empty. So, page table is [0,1,2,3] mapped to [2,7,1,5] and the instr will load the physical memory on location 1106 (while user program only sees logical memory of 2106).

FORMULA. Phys-addr = Frame_number · sizeof(physical_frame) + offset [offset := displacement from beginning of physical frame]. Essential tricks: two impt design decisions to simplify addr translation: {1. Keep frame-size/page-size as power-of-2; 2. Phys-frame-size == Logic-frame-size}. Illustration is Logic-addr is $[po]$, p has $m-n$ bits and o has n bits, and translation mechanism maps p to f (frame number) while keeping offset unchanged. Conclusion of Addr-Translation Formula: Given page/frame size of 2^n and m bits of logical addr; for Logic-addr LA, set p := most significant $m-n$ bits of LA, o := remaining n bits of LA; use p to find frame-number f (from mapping mechanism like page-table); and finally, $PA := f \cdot 2^n + o$.

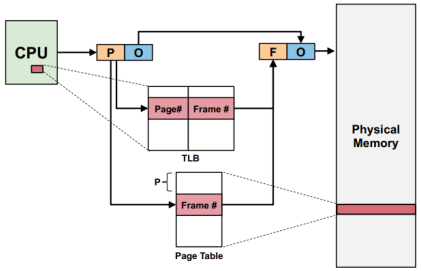
The above paragraph is a rephrasing of the next.

{begin redundant section} Page table: to support translation, we will use a page table, which is an array where {index: page number, value: frame number}. Two tricks are employed in calculation: {keep frame/page size as (a) power of 2, keep frame and page size equal}. Then, for page/frame size of 2^n and address of m bits, to translate, we copy last n bits + offset + over, then for (next????) rightmost $m-n$ (page nmber), we index it into page table and replace it with the value (frame number). {end redundant section}

Analysis of paging: {No external fragmentation (NOT POSSIBLE): no leftover physical mem-reg (every single free frame can be used); has internal frag (YES, BUT INSIGNIFICANT): when a logical mem-space is not a multiple of page size (max one page per process is not fully utilized); clear separation of logical and physical address-space: flexibility and simple translation}.

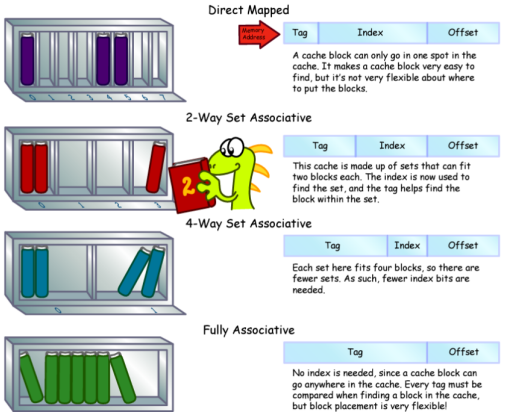
(A Pure-software) Implementation: each proc has its own page table, stored in its PCB, this PCB is in RAM. [Improved Understanding] So, this (page table) is included in the memory context of a process (or pointers to page tables; today's page tables are quite big). However, this means we require 2 RAM accesses for each mem-reference: the two accesses are {1. read the indexed page table entry to get frame number; 2. access to actual mem-item}.

Translation Look-Aside Buffer



Description: bottom is 2 lookups (through page-table), but top is through TLB ("page-table entries cache").

RECAP (2100 or look online in this link: https://csillustrated.berkeley.edu/PDFs/handouts/cache-3-associativity-handout.pdf) about different types of caches. The 3 types here:



Translation Look-Aside Buffer. The TLB is on the chip, and provides hardware support for paging. Caches 4KB of page table entries (??? what does this mean ??? Done: check if have; if have then hit; if miss then take extra penalty to go through entire "normal uncached" run-through) and can cache multiple (table entries) at once; like associative cache (have to look this up again later, what does this mean; Done: see above). Properties/Behaviors: {Fast: takes 1 ns, vs the 50 ns for RAM; TLB-Hit vs TLB-Miss: RAM is only accessed upon the latter, and TLB is updated after that.

TLB's Impact on Mem-Access-Time: AMAT (average-mem-access-time) is = $TLBhit + TLBmiss$. Say hit rate is 90%, this equals $P(TLBhit) \cdot \text{latency}(TLBhit)$ plus same thing for TLB miss. This equals (assuming 1 ns, 50 ns) equals $90\% \cdot (1ns + 50ns) = 10\% \cdot (1ns + 2 \cdot 50ns)$. Compare with no TLB: memory access two time (100 ns). Notes: latency of filling in TLB entry can be hidden by the hardware; this impact of caches is ignored.

Context switching; what happens to PTBR (Page-table Base Register): There are 3 things to note in context/process switch: {Page table: we change the pointer in a special register called "base register" from P to Q's base; Physical frames: OS need correction schemes or protection mechanisms so newly running proc Q not interfere with everything else; TLB: This is a global object. In CS2106, can assume TLB is "fully flushed" (???), as it is a part of a proc's hardware context. What does this mean? It means that if not flush, we CAN have incorrect translation (moreover, we flush to avoid correctness, security and safety issues), This flushing ("emptying" whatever was inside that belonged to previous proc) makes each proc encounter initial latency in the form of a couple of TLB misses, until its TLB "fills up".}

Protection. We can extend the paging entries to include bits to support mem-protection. Properties: {(To facilitate) Access(ing the)-right bits (a note: right here does not mean "opposite of left", right here means "privilege/power (Indonesian: hak)": on whether the page itself is writable, readable or executable, e.g. you cannot write over the text of the proc, but you can execute or read OR you cannot execute data of the proc, but can read and write; Valid bit ("OS as security guard"): some pages may be out of range for certain processes for certain reasons (OS will set these valid bits when running). If out-of-range access is done, OS will catch.} BOTH is checked in HARDWARE; and out-of-range accesses will trigger some interrupt and caught by OS to be passed/handled by some interrupt handler.

Page Sharing. We can naturally now have mutliple pages pointing to the same physical frame (??? actually yes, very magical, like pipelining but with memories and their accesses; which if not shared, will result in efficient memory usage because the physical mem-space will have multiple copies of same code across multiple processes). For example, when some library code are shared between processes. We can use a SHARED BIT in the page table entry to track whether the page is shared, OR when parent forks a child. Properties: {Shared code page: as mentioned, when there's library code, syscalls, etc; Copy-on-write: when a parent forks, the parent and child share same pages; i.e. page table is copied, but frames remain (why this happen, and what does "frame remains" mean?) This means that initially (after forking), every frame is pointing to the same frames. Using shared bit, when the child needs to update a shared page (AND only when a child *writes*), then the frame is dup-ed and page "unshared" (seems like lazy writing; only dup-ing when it is necessary). When any process tries to write (regardless of when it's parent or child), that is the moment when we make a copy and make corresponding changes. Note: this is why fork is fast! The pages are shared between multiple processes}.

Segmentation. Mem-space usually contains multiple reg(ion)-s, all with different usages in a proc. Some remain const in size (i.e. data and text), some will shrink and grow during execution time (i.e. stack, heap, library code regions). It is thus difficult to put all the regions together in a contiguous (logical) mem-space AND allow them to shrink/grow freely. Also hard to check if a mem-access is in-range.

Segmentation Scheme. Basic Idea: in Paging, mem-space considered "a single entity". We don't consider them as 4 distinct regions, we just "blindly allocate". The Motivation for this scheme is then some regions may shrink/grow during execution time. This is hard to achieve is when process uses 1 piece of memory space; e.g. at compile time, not know how much the stack (of a particular process) may/is-going-to-grow.

This scheme turns the mem-space into "a collection of segments" where each segment is mapped into contiguous physical parts (of same size). Each mem-segm has name and limit.

Consist of 3 things (?)

1. MEMORY SEGMENTS: split the regions into their own segments; {name: for reference, usually translated to an index, e.g. {Text = 0; Data = 1; Heap = 2}; base: physical base address; limit: indicate range of segment}.

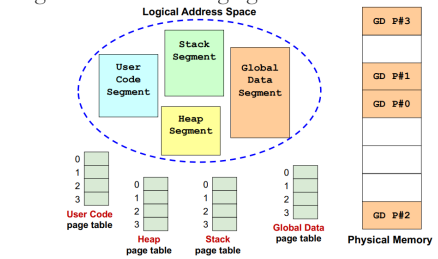
2. SEGMENT TABLE: we keep a table of [Base, Limit], indexed by the name indices/segment ids (so, in my mental space, I imagine it as segtable[index] := pair({Base, Limit})). The algorithmic "execution" of a segtable: {1. With [SegId, Offset], we use SegId to get the [Base, Limit]; 2. We check if Offset < Limit, if so, seg(mentation) fault; 3. Else we access base + offset}. We store this seg-table inside registers since its size is fixed and small, AND it's (registers are) fast. The motivation for this is the segtable is equal to # of proc-s (i.e. *Segtable <<<< Page Table*).

3. ANALYSIS OF SEGMENTATION: {Independent segments: each segment is contiguous and independent, and can shrink and grow independently (i.e. not enough memory? Can just find another pace if not enough 4head!); (Unavoidable) external frag: variable size is contiguous mem-reg-s, resulting in the same problem as always; Not-the-same-as-paging: solving different problems}.

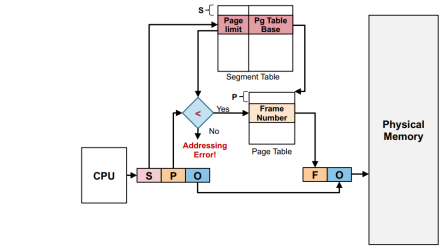
Segmentation with Paging (Hybrid). Lecture says: "Most commonly used. The intuitive next step is to combine the 2".

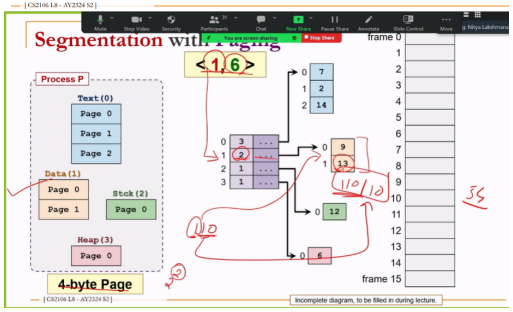
Each segment is now composed of pages and has a page table of its own (can be of different sizes). The seg-table now points to page-table address instead of base address. Page limit remains unchanged.

Segmentation with Paging: **Basic Idea**



Segmentation with Paging: **Illustration**





Note: example solution is $1101(10)_2$ or **54**.

9 Virtual Memory Management

Motivation: removing assumption #2. 2 reasons: {What if logical mem-space of proc is >> than physical memory?; What if same program is executed on a computer with less phys-mem?}

Basic Idea. Key observations: {1. Secondary Storage Capacity (HDD/hard-disk-drives, SDD (? I think it's SSD or solid-state-drives)) >> Physical Memory Capacity; 2. Some pages are accessed much more often than others}.

So the basic idea is split logical address space into small chunks: {Some chunks reside in physical mem; other are stored on SECONDARY STORAGE}. The most popular approach: extension of paging scheme: {Logical mem-space split into fixed-size-page; some pages may be in phys-mem, others in sec-storage}.

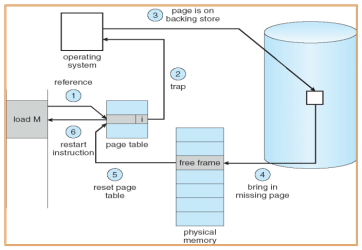
Illustration: Logical Memory Space (== Virtual-Mem-Space) = Physical Memory + Secondary Storage (extension of basic paging scheme, i.e. the "virtual" part). "Gives the illusion of having more memory than I (actually) have". Call the swap (primary/physical memory is called "frame" as per usual) space in Linux Virtual-Memory installation.

Extended Paging Scheme. Basic idea remain unchanged: use PAGE-TABLE for Virtual/Logical → Physical address translation.

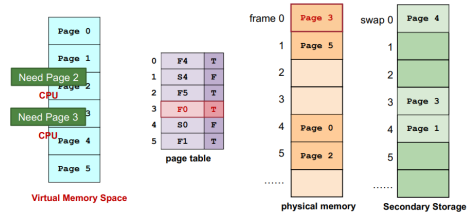
New addition: 2 page types: {Memory resident (pages in phys-mem); Non-memory resident (pages in sec-stor)}, we use RESIDENT BIT in page-table entry (indicates whether page is resident (in mem)). CPU can only access mem-resident pages: {Page Fault: when CPU tries to access non-mem res-page; So, OS needs to bring a non-mem res-page into phys-mem}.

Accessing Page X: General (6) Steps. First step (DONE BY HARDWARE): check page table "is page X mem-resident?", {Yes: access phys-mem-loc. Done; No: raise an exception!} Next (5) steps: {2. Page Fault, OS takes control; (3-6 DONE BY OS) 3. Locate page X in secondary storage; 4. Load page X into a phys-mem; 5. Update page table; 6. Go to step 1 to re-execute the SAME instruction, this time with the page in memory.}

Virtual Memory Accessing: Illustration



Page Fault: Illustration

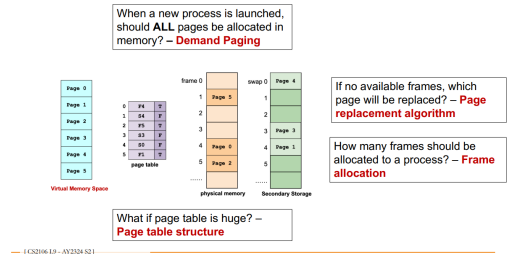


10 The Week 9 Lecture (i.e. L9b)

The previous part was for an introduction that was on Week-8 Lecture ending (only touched for 10 minutes).

Issues of Virt-Mem. 3 issues: {1. Elephant in the room: Secondary-Storage access time » Physical memory access time. ~ 5 orders of magnitude: milliseconds » nanoseconds; 2 (still related to 1). If mem-access results in page-fault most of the time (i.e. to load non-resident pages into memory), the entire system CAN SLOW DOWN SIGNIFICANTLY (known as thrashing); 2'. How to ensure thrashing not happen? (related to 2': how we know after page is loaded into mem, that it's likely to be useful for future accesses)}

Overview



The Answer is Locality. (we are still recapping here) Unlikely that page-faults occur repeatedly) to occur because locality. What is locality? Most programs exhibit these behaviors: {1. Most time are spent on relatively small part of code; 2. In any given time period, accesses are made to relatively SMALL part of data.} Properties of locality:{Temporal: mem-addr has-been-used is likely to be-used-again; Spatial: nearby mem-addr-s are likely to be-used-next. They are included in 1 page.}

Virt-Mem + Loc-Princ. Both forms of locality help amortize the cost of loading the page (may be needed again after loaded, and a page that contains (many) consecutive locations that are likely to be used in future [so later access less page-fault-s]). Exceptions to this amortization: poorly-designed or maliciously-intentioned programs (i.e. those that exhibit bad behavior).

Still Recap: why useful? {Fully separates logic-mem from phys-mem: amount of phys-mem no longer restrict size of log-mem-addr; more efficient use of phys-mem (not-needed-pages can be on sec-stor); allows more processes to reside in memory (↑ CPU utilization as ↑ processes being able to run}.

Intro: What happens if we don't do demand-paging? IF all text/instruction + data be allocated in memory, considerations: {large startup cost if there are large num of pages to init+alloc (how to pick, if we don't load everything?); need to reduce footprint of proc-s in phys-mem SO THAT more proc-s can use mem}.

Demand Paging. OS starts with no memory-resident page only copies a page into mem IF a page fault occurs (AS OPPOSED/COMPARED TO anticipatory paging). This makes it so that un-needed pages will almost-never be loaded. Pros: {more efficient use of phys-mem; fast startup time for new process}. Cons: {proc-s may appear sluggish at start due to multiple page-faults; and those page faults may have cascading effects on other proc-s (i.e. thrashing); with large num of pages are on disk, the page tables themselves still take up a lot of space in mem; results in high overload and fragmentation (since table itself needs to occupy several pages)}.

BIG-SECTION 1. Page Table Structures. How to structure page-table for EFFICIENCY? (much bigger problem than you might realize; since we have HUGE PAGE TABLE with LARGE LOGICAL-MEMORY-SPACE).

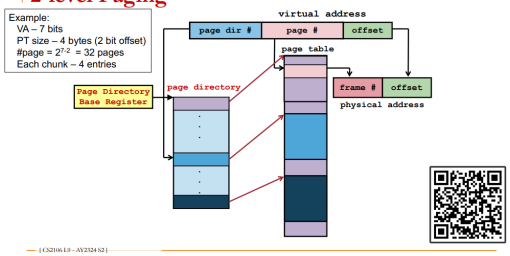
IF we do direct-paging. Keep all entries in single page-table, and we allocate each proc this huge page table. HOWEVER, current modern-computer-systems provide huge logic-mem-space. (Current) Situation: {4GB (32-bit) was norm in past, 8TB+ is possible now; huge logic-mem-space ⇒ huge num of pages ;(indirect cause-and-effect) → (IF) each page has a page-table-entry ⇒ huge page table}. Problems (with huge page table): {high overhead, page-table alloc (page-table can occupy several frames)}.

Examples. 1: mid 1980s. V-Addr-s: 32-bits, Page-size: 4KB, $p = 32 - 12 = 20$ (the 12 is a fixed number; since $4\text{KB} = 2^2 \cdot 2^{10}$ bytes. We divide entire logical-mem-space of 2^{32} bits into 2^{12}-sized-pages. Can always find # of pages by dividing entire space by size-of-each-page). Since the "residue" p bits can (and is also an upper bound) be used to specify one unique page, so there are 2^{20} allowed page(-table-entry)s/PTE-s. So P-tab-size is $2^{20} \cdot 2\text{B} = 2\text{MB}$ (why 2 bytes per entry? 1 byte is not enough since there is extra info besides just frame no; e.g. valid bit, access-right-bits etc; nowadays, range from 4-8 bytes, more often, we see 8-byte p-table entries).

2. Today's midrange laptop: V-Addr-s: 64-bits ($16 * 1024\text{GB}$ of V-Addr-space!), Page-size: 4KB (12 bits for offset, as in mid-1980s model), Physical memory 16GB with P-Addr-s = 4+10+10+10 = 34-bit. How many V-pages: $2^{64}/2^{12}$ PTE entries, How many P-pages: $2^{34}/2^{12}$ PTE entries, in reality, PTE size is 8B (with other flags). So, p-table-size: $2^{52} * 8\text{B} = 2^{55}\text{B}$ per process! Compare with phys-mem-size 2^{34}B.

// TODO: revise this, especially minute 52 of L9b!

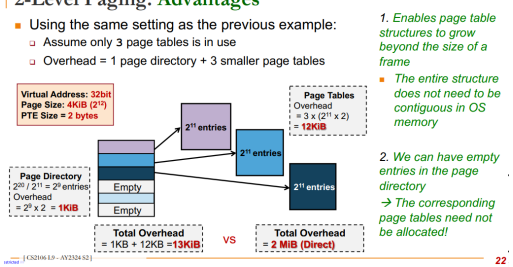
2-level Paging



2-Level-Paging! Observation: process may not use entire v-mem-space; so full-page-table is a waste! Idea: page the page table:{Split whole page table into regions, only a few regions are used (as mem-usage gros, new region can be alloc-ed); this is similar to splitting logic-mem-space into pages; AND we need a "directory" to keep track of the regions (analogue to the relationship between page-tables and (real/actual) data-pages}.

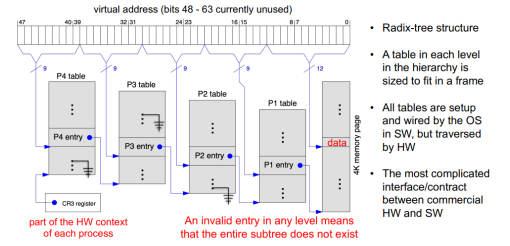
Description: {split page-table into smaller page-tables, each with a page-table-num; if original page-table has 2^p entries: {with 2^m smaller page-tables, m bits needed to uniquely identify (one page-table), each smaller page-table contains 2^{p-m} entries}; to keep track of smaller page-tables: {a single page-directory is needed; page-dir contains 2^m indices to locate each of smaller page-table}}.

2-Level Paging: Advantages



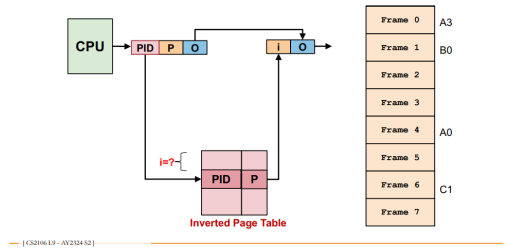
Disadvantages/Problems. Requires TWO serialized memory access JUST to get frame number (one access for directory, another for page-table, and only then can access data; and that doesn't count if page-table is on virtual-mem-space — this is possible by tutorial question). TLB-as-a-solution? Since it eliminate page-table accesses? HOWEVER, TLB-miss-es experience longer page-table-walks (traversal of page-tables in hardware).

Hierarchical Page Table – Todays' Midrange Laptop (*Often in Exams*)



Inverted Page-Table. What the heck is this?

Inverted Table: Illustration



Basic Idea: {p-table is a per-process info (if M proc-s in mem, there are M indep p-tables); observations: {on-

ly N mem-frames can be occupied, and out of M page tables, only N are valid; so it's a huge waste, sincd $N << M$ (page tables) idea: keep single mapping of [phys-frame → (pid, page#)], pid = proc-id, page# logic-page-num corresponding to the proc (note: page# not unique among proc-s, another-note: pid + page # can uniquely identify a mem-page)}.

Lookup: {Normal-page-table: PTE-s ordered by page-num (lookup page X: access X^{th} entry); inv-page-table: PTE-s ordered by frame-num (lookup page X: need search whole table)}. Pros/Cons: {Adv: huge saving (one table for all (maybe insanely many) proc-s); Dis: slow translation}. Application: in practice, inv-tables often used as "auxiliary structure" (e.g. to answer questions like: who are (all) sharers of phys-frame X?).

BIG-SECTION 2: Page-Repl Algos. Which page shld be replaced (evicted) when needed (i.e. when a page-fault occurs)? When a page is evicted: {if clean(-page) (i.e. not-modified) → no need to write-back; else dirty(-page) (i.e. modified) → need to write-back}. Algos to find a suitable repl-page: OPT, FIFO, LRU, Second-Chance (clock) [only talk abt these 4 in CS2106]. Modelling mem-ref-s: in actual mem-ref [logic-addr = page-num + offset]; however, for studying (page-repl-algo) purposes, only page-num is important. Therefore, to simplify discussion, mem-ref-s are often modeled as mem-ref-strings (i.e. a sequence of page-num-s).

Evaluations of Page-Repl-Algos. Mem-access time $T_{\text{access}} = (1-p)*T_{\text{mem}} + p*T_{\text{page-fault}}$, with p is probability of page-fault (others are self-explanatory; T stands for (access) time). Since $T_{\text{page-fault}} \gg T_{\text{mem}}$, need to reduce p to keep T_{access} reasonable. (Good algo should minimize total-num of page-faults + be fast!)

1. OPT (optimal page repl). General idea:{replaces page that WILL NOT be needed again for the LONGEST PERIOD of time; guarantees minimum num of page-faults (try to prove it! Proof sketch: if replace a more-upcoming-page that is to-be-used in near(er)-future, additional page-faults can occur)}

Not realizable: need future knowledge of mem-ref-s. Still useful: {as a base of comparison for other algo-s; the closer to OPT, the better the algorithm}.

OPT

Time	Memory Reference	Frame A	B	C	Next Use Time			Fault?
1	2	2			3	∞	∞	Y
2	3	2	3		3	9	∞	Y
3	2	2	3		6	9	∞	
4	1	2	3	1	6	9	∞	Y
5	5	2	3	5	6	9	8	Y
6	2	2	3	5	10	9	8	
7	4	4	3	5	∞	9	8	Y
8	5	4	3	5	∞	9	11	
9	3	4	3	5	∞	∞	11	
10	2	2	3	5	12	∞	11	Y
11	5	2	3	5	∞	∞	11	
12	2	2	3	5	∞	∞	∞	

2. FIFO (pgae repl). General idea: mem-pages evicted based on their load-time → evict oldest mem-page. Imple: OS maintains queue of resident page-num-s. Details of imple: {remove first page in queue if repl is needed; update queue during page-fault-trap}. This is imple to imple (i.e. to hardware support needed).

FIFO

Time	Memory Reference	Frame A	B	C	Loaded at Time			Fault?
1	2	2			1			Y
2	3	2	3		1	2		Y
3	2	2	3		1	2		
4	1	2	3	1	1	2	4	Y
5	5	5	3	1	5	2	4	Y
6	2	5	2	1	5	6	4	Y
7	4	5	2	4	5	6	7	Y
8	5	5	2	4	5	6	7	
9	3	3	2	4	9	6	7	Y
10	2	3	2	4	9	6	7	
11	5	3	5	4	9	11	7	Y
12	2	3	5	2	9	11	12	Y

Problems with FIFO. Belady's anomaly (exhibits unintuitive behavior: ↑ frames → ↑ page-fault-s. Intuition for good algo: ↑ phys-frames (i.e. more RAM), should be ↓ page-faults). Example: mem-access {1,2,3,4,1,2,5,1,2,3,4,5}; 3 frames gives 9 PFs, 4 frames gives 10 PFs. So, bad performance in practice.

3. LRU (least-recently-used). General idea: makes use of temporal locality: {honor the notion of recency; replace the page that hasn't been used in longest time ("least recent" one)}. Another way to put this assumption/filtration-algo is "we expect a page to be reused in a short time window" (practical temporal-locality); so the contrapositive is "have not been used for some time → most likely will not be used again".

Notes: this algo aims to approximate the OPT algo: {predicts future by mirroring the past; gives good results in practice (does not suffer from Belady's anomaly)}.

LRU

Time	Memory Reference	Frame A	B	C	Last Use Time			Fault?
1	2	2			1			Y
2	3	2	3		1	2		Y
3	2	2	3		3	2		
4	1	2	3	1	3	2	4	Y
5	5	2	5	1	3	5	4	Y
6	2	2	5	1	6	5	4	
7	4	2	5	4	6	5	7	Y
8	5	2	5	4	6	8	7	
9	3	3	5	4	9	8	7	Y
10	2	3	5	2	9	8	10	Y
11	5	3	5	2	9	11	10	
12	2	3	5	2	9	11	12	

LRU imple details: not easy, need to keep track of "last access time" somehow. Hardware support: 2 options.

A. Use a Counter. Details:{Logical "time" counters, which is incremented for every mem-ref; page-table-entry has a "time-of-use" field (store time-ctr value whenever it is referenced); we replace the page with smallest "time-of-use"}. Problems: {need to search through all pages; "time-of-use is forever increasing (possible overflow!)}.

B. Use a "Stack". Details: {maintain a stack of page-num-s; if page X is ref-ed: remove from stack (for existing entry) THEN push on top of stack; THEN replace the page at BOTTOM of STACK (no need to search through all entries, for this step!)}. Problems: {not a pure stack (entries can be removed from anywhere in the stack); hard to imple in hardware}.

CLOCK

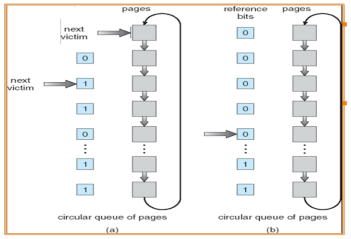
Time	Memory Reference	Frame (with Ref Bit) A	B	C	Fault?
1	2	▶2 (0)			Y
2	3	▶2 (0)	3 (0)		Y
3	2	▶2 (1)	3 (0)		
4	1	▶2 (1)	3 (0)	1 (0)	Y
5	5	2 (0)	5 (0)	▶1 (0)	Y
6	2	2 (1)	5 (0)	▶1 (0)	
7	4	▶2 (1)	5 (0)	4 (0)	Y
8	5	▶2 (1)	5 (1)	4 (0)	
9	3	▶2 (0)	5 (0)	3 (0)	Y
10	2	▶2 (1)	5 (0)	3 (0)	
11	5	▶2 (1)	5 (1)	3 (0)	
12	2	▶2 (1)	5 (1)	3 (0)	

▶ Victim Page

Second-Chance Page Repl (a.k.a. CLOCK). General Idea: {Modified FIFO, to give a second chance to pages THAT ARE ACCESSED; each PTE now maintains a ref-bit: {1: accessed since last reset; 0: not accessed}}.

Algo: {1. Oldest FIFO page is selected (victim page); 2. if ref-bit == 0 → page is repl-ed. Done, algo exits; 3. If ref-bit == 1 → page is skipped (i.e. given a 2nd chance): {ref-bit cleared to 0; effectively resets arrival time → page taken as newly-loaded' next FIFO page (victim) is selected (i.e. go to Step 2 but for next page after this)}}. Note: {this algo degenerate into FIFO algo when all pages has ref-bit == 1; adds some notion of recency AND works well in practice}.

Second-Chance: Implementation Details



45

BIG-SECTION 3: Frame Allocation. How to distribute limited phys-mem-frames among proc-s? (Assume, for notation, N phys-mem0frames, M proc-s competing) Simple: equal alloc (N/M frames) or proportional alloc (each proc $p$get $\text{size}_p/\text{size}_{\text{total}} * N$ frames).

Assumption lifting. Implicit assumption for page-replo-algo-s so far: victim pages ARE SELECTED AMONG PAGES OF THE PROCESS that causes page-fault (local replacement). If victim page can be chosen AMONG ALL PHYS-FRAMES, proc P can take (a) frame from proc Q, evicting Q's frame during replacement (global replacement).

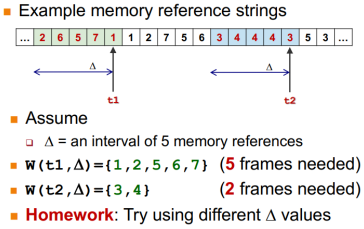
Pros/Cons: {Local:{Pros: num of frames alloc to a proc remains constant → performance stable between multiple (re)runs; Cons: if alloc-ed frames not enough → hinder progress of a proc}; Global:{Pros: allow self-judgement betwen proc-s, proc needs more frame can get from others that need less; Cons: badly behaved proc-s steal frames from other proc + num-frames alloc-ed to proc can be different on rerun}}.

Thrashing, in relation to Local/Global Repl. If thrashing, need heavy I/O to bring non-resident pages into RAM (to be resident). It's hard to find right # of frames. If global is used: {thrashing proc "steals" page from other proc; causes other proc to thrash (cascading thrashing)}. If local is used: {thrashing can be limited to that (one) proc; BUT that proc can use up I/O bandwidth + degrade performance of other (affected) proc-s}.

The observation that "set of pages referenced by a proc is relatively constant in a (particular) period of time" gives the defn of a working set. However, as time passes, this set can change (different prog phases require different data). Example: function is executing (references likely on local variables that define function's locality) and terminates (ref-s change to another set of pages (abruptly!!!)).

Working Set Model: monitors mem-usage patterns of proc-s and adjusts frame-alloc-s based on working-set of each proc. Define work-set-window Δ as a particular interval of time (say, interval of 5 mem-ref-s), and $W(t,\Delta)$:= active pages in the interval at time t. The algo allocates enough grames for pages in $W(t,\Delta)$ to reduce possibility of page-fault.

Working Set Model: Illustration



Summary. Covered virt-mem (why it's needed + how it works) and 3 BIG-SECTIONS (aspects) of virt-mem-management: {using different p-table-structure to reduce p-table-overhead; using diff p-repl-algo-s to reduce p-fault; how frame-alloc affects p-fault of a proc}.

11 Intro to File Systems

OVERVIEW. File System: {defn; vs-mem-management; motivation}, File: {metadata, operations}, Directory: {basic, structure}.

File-System Motivation: {(why need file system?) Phys-mem is volatile, use external storage to store PERSISTENT info; (why need OS to (help) manage file system?) direct-access to storage media is not portable, dependent on hardware specs and organization (i.e. phys-mem in one system not portable to another (differently built) system); (So, OS helps the) f-sys (to) provides: {abstraction on top of physical media; high-level resource-management scheme; protection between proc-s and users; sharing between proc-s and users}}.

File-System General Criteria: {Self-contained: {info stored on a media is enough to describe the entire organization; should be able to "plug-and-play"}; persistent: beyond the lifetime of OS and proc-s; efficient: {provides good management of free + used space; minimum overhead for bookkeeping info}}.

Memory Management vs File Management

	Memory Management	File System Management
Underlying Storage	RAM	Disk
Access Speed	Constant	Variable disk I/O time
Unit of Addressing	Physical memory address	Disk sector
Usage	Address space for process **Implicit** when process runs	Non-volatile data **Explicit** access
Organization	**Paging/Segmentation:** determined by HW & OS	**Many** different FS: ext* (Linux), FAT* (Windows), HFS* (Mac OS) etc.

Note: disk consists of several circles, so accessing data in a inner-circle-d sector vs outer-circle-d sector will take different amount of time. Another note: when we use program, we have "no choice" but to invoke RAM (implicit usage) (since RAM is necessary addr-space); if we want to use data from stored memory (i.e. file), need to explicitly declare our (intention to) use.

Key Topics. F-sys abstraction (L10): discuss logical entities present in f-sys, e.g. files/directories. F-sys imple approaches (L11): {f-sys layout; common imple schemes; discuss pros + cons (of each)}. Additional: F-sys case studies (L12) {THIS IS ADDED ARGH, SO I WON'T HAVE TOO MUCH TIME to study other materials}.

F-sys abstractions. "You mean files and folders ARE NOT REAL?"

File Basic Definition. Represents a logical unit of info created by process. An abstraction (essentially an Abstract-Data-Type); files are (+ contains) a set of common operations with various possible imple. Contains (i) data: info structured in some ways; (ii) metadata: additional info associated with the file (also known as file attributes).

File Metadata

Name:	A human readable reference to the file
Identifier:	A unique id for the file used internally by FS
Type:	Indicate different type of files E.g. executable, text file, object file, directory etc
Size:	Current size of file (in bytes, words or blocks)
Protection:	Access permissions, can be classified as reading, writing and execution rights
Time, date and owner information:	Creation, last modification time, owner id etc
Table of content:	Information for the FS to determine how to access the file

File Name (human-readable identity of the file). Different FS has different naming-rule (to determine valid f-name). Common naming rule: {length of f-name; case sensitive-ity; allowed special symbols; file extension (usual form name.extension, on some FS, extension is used to indicate f-type)}.

File Type. OS commonly supports a num of f-types. Each f-type has {associated set-of-oper-s + (possibly) a specific-prog for processing}. Common f-types: {reg-files (contains user info (ASCII + binary files); directories (sys-files for FS structure); special files (char/block-oriented)}.

(major types of regular files): {ASCII-files (e.g. txt-f-s, programming source-codes, etc). Can be displayed/printed as-is. Binary-files (e.g. executable, Java class-f, pdf file, mp3/4, png/jpeg/bmp etc). Have a predefined internal-structure that CAN BE PROCESSED by SPECIFIC PROGRAM (JVM to execute Java-class-f, PDF-reader for pdf-f, etc)}. How to Distinguish between f-type-s? {use f-ext as indication (windows OS); use embedded-info in file (Unix; usually stored at the beginning of the file, known as "magic number")}.

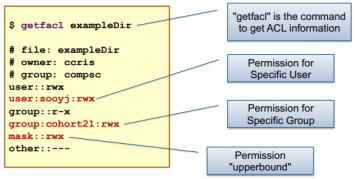
Operations on F-metadata: {rename (change f-name); change attr-s (f-access-permissions, dates, ownership, etc); read-attr (get f-creation-time)}.

File Protection. Ctrl-ed access to info stored in a file. Type-of-access: {r (retrieve info from f);w (write/rewrite f); x (load f into mem and exec it); a(ppend) (add new info to EOF); del (remove file from FS); list (read metadata of a f)}. How? {Most common approach: restrict access based on user-identity; most general scheme is ACL: {a list of user-identity + allowed access-types; pros: very customizable; cons: too much info associated with f}; a common condensed f-protection scheme is permission-bits}.

What is "perm-b-s"? Classify users into 3 classes: {Owner (user who created f); group (set of users who need similar access to a file); universe (all other users in system)}. Unix: define permission of 3 access types (r,w,x) for the 3 classes of users.

File Protection: Access Control List

- In Unix, Access Control List (ACL) can be:
  - Minimal ACL (the same as the permission bits)
  - Extended ACL (added **named users / group**)



File Data. Structure (possibilities): {Arr-of-bytes (each byte has unique offset from f-start. We talk about offset to reach the n^{th} byte here); fixed-len-records: {array of records, can grow/shrink; can jump to any record easily (offset of n^{th} record is size-of-rec $*(n-1)$)}; var-len-records (flexible but harder to locate a record)}. Access Methods: {sequential-access: data read in order (can't skip but can be rewound); random access: data can be read in any order. Can be provided in 2 ways: {$read(offset)$: every read operation explicitly state the position to be acc-ed; $seek(offset)$: a special operation, provided TO MOVE to a new loc in f}; direct-access: used for f that contains fixed-len-records. Allows random access to ANY record DIRECTLY. Very useful where there is a large amount of rec-s. Basic random access method can be viewed as a special case (of this method), where "each-record == one byte"}. Unix and Windows use random access (I think)

From google it says: "Windows and Linux don't differentiate sequential and random access files anymore than the CPU differentiates byte and character values in memory; it's up to your application to treat the files as sequential or random access."

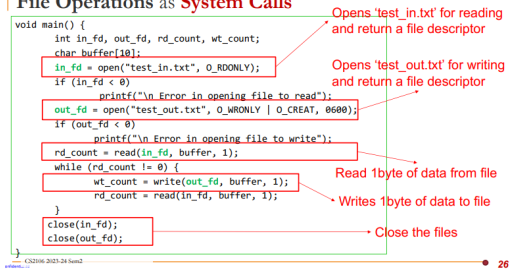
File Data: Generic Operations

Create:	New file is created with no data
Open:	Performed before further operations To prepare the necessary information for file operations later
Read:	Read data from file, usually starting from current position
Write:	Write data to file, usually starting from current position
Repositioning:	Also known as seek Move the current position to a new location No actual Read/Write is performed
Truncate:	Removes data between specified position to end of file

File Operations. As syscalls, OS provides f-oper-s as syscalls (provide protection, concurrent, and efficient access). File-related Unix-syscalls are in last part of L10 notes (in this latex file).

Unix-syscalls Header Files: *#include <sys/types.h>*, *#include <sys/stat.h>*, *#include <fcntl.h>*. General info: {opened-f has an id (file-desc, which is an integer). This is used for other operations; file is acc-ed on a byte-to-byte basis (no interpretation of data)}.

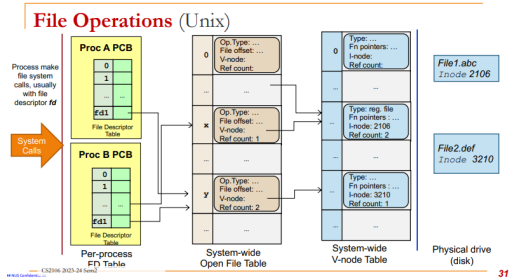
File Operations as System Calls



File Information in the OS. Info (that is) kept for an opened-file: {F-ptr (keep track of current position within a file); f-descriptor (unique identifier of the f); disk location (actual f loc on disk); open count/reference count (how many proc has this f opened?), this is useful to determine when to remove entry in table}. Since several proc-s can open same file + several diff files can be opened at any time, the COMMON APPROACH is to USE 3 TABLES: {per-process open-file table: {to keep track of open files for a proc, each entry points to the system-wide open-file table entries}; system-wide open-file table: {to keep track of all open files in the system, each entry points to a V-node entry}; system-wide v-node (virtual-node) table:{to link with the f on phys-drive; contains info about phys-loc of the f.}}.

File Operations and Process (that) Shares(ing) Files in Unix. Case 1: file is opened twice from 2 proc-s: I/O can occur at independent offsets. Case 2: 2 proc-s using same open-file table entry: only 1 offset → I/O changes the offset for the other process.

File Operations (Unix)



Directory. Basics: {Dir/folder is used to: {provide logical grouping of files (user view of directory); keep track of files (actual system usage of directory)}; there are several ways to structure directory (to be exact, 4 is discussed)}.

Single-Level. Self-explanatory.

Tree-Structure(d). General-Idea: {directories can be recursively embedded in other dir-s; naturally forms a tree structure}. 2 ways to refer to a file: {Absolute Pathname. Dir-names followed from root-of-tree TO final-file; Relative Pathname. Dir-names followed from current-working-dir (CWD), CWD can be set-explicitly OR implicitly changed by moving into a new directory (under shell prompt)}.

DAG. If a file CAN be shared: {only 1 copy of actual content; "appears" in multiple dir-s (with different pathnames); 2 file names referring to same f content}, then tree structure → (becomes) DAG. Imple in Unix: hard-link not allowed for directories.

What is Unix Hard-Link (for DAG)? Consider: dir A is owner of file F + dir B wants to share F. Hard-link: {A and B has separate ptr-s point to actual file F in disk; pros: low overhead, only ptr-s are added in dir; cons: deletion problems (e.g. if B deletes F? If A deletes F?)}. Unix Command: "*ln*".

General Graph. General-Graph dir structure is NOT DESIRABLE: {hard to traverse (need to prevent infinite looping); hard to determine when to remove a f/dir}. In Unix: symbolic link IS ALLOWED to link to directory, so general graph CAN BE CREATED.

What is Unix Symbolic Link/Soft Link? {Symbolic link is a special link file, G, where G contains path name of F. When G is accessed, find out where is F THEN access F; pros: simple deletion, {if symbolic link G deleted: G deleted, not F; if linked file is deleted, F gone, G remains (but not working, i.e. dangling link}; cons: larger overhead, special link file take up actual disk space}. Unix Command: "$ln -s$".

Summary: covered basics of f-sys from a user POV, understand basic req-s of a f-sys, understand components of a FS (that is, file and directory).

UNIX Context. *open*() takes in *char**path and *int* flags (many options can be set; using bit-wise-OR: {read, write, or read+write mode; truncation, append mode; create file (if not exists); many, many more :)}). Returns file descriptor (fd) integer (−1 failed to open file, ≥ 0: unique index for opened file). By convention: default fd-s: {STDIN (0); STDOUT (1); STDERR (2)}.

Open existing file for read only: $fd = open($ "*data.txt*", O_RDONLY); Create file if not found, open for rd+wr: $fd = open($ "*data.txt*", $O\_RDWR \mid O\_CREAT$);

read(): {fn-call: *int* read (*int* fd, *void**buf, *int* n); purpose: reads up to n bytes from curr-offset into buffer buf; returns num of bytes read (can be $0 \ldots n$), if $< n$: EOF is reached; param-s: {fd: file-descriptor (must be

opened-for-read); *buf*: an array "large-enough" to store *n* bytes}; *read*() is sequential read, starts at curr-offset and increments offet by bytes read}.

write(): {fn-call: *int* write (*int* fd, *void**buf, *int* n); purpose: writes up to *n* bytes from curr-offset into buffer *buf*; returns num of bytes read (can be 0 ... *n*), otherwise −1: error; param-s: {*fd*: file-descriptor (must be opened-for-write); *buf*: an array at least *n* bytes with values to be written}; POSSIBLE ERRORS: {exceeds file-size limit; quota; disk space; etc}; *write*() is sequential write: {starts at curr-offset and increments offet by bytes written; CAN increase file size beyond EOF (by appending new data}}.

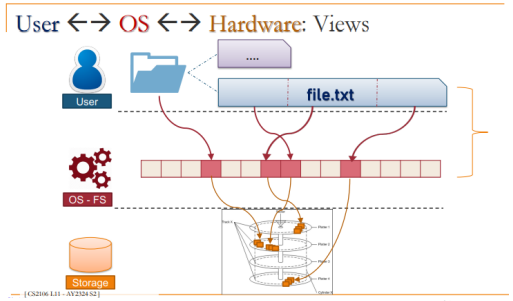
lseek(): {fn-call: *off_t* lseek (*int* fd, *off_t**offset, *int* whence); purpose: move current position in file by *offset*; returns current offset in file (≥ 0), otherwise −1: error; param-s: {*fd*: file-descriptor (must be opened); *offset*: positive = move forward, negative = move backward; *whence*: point-of-reference for interpreting the *offset*: {SEEK_SET: absolute offset (count from file start); SEEK_CUR: relative offset, from curr-position (+/-); SEEK_END: relative offset from EOF (+/-)}}; can seek anywhere in file, even beyond end of existing data}.

close(): {fn-call: *int* close (*int* fd); returns 0: successful, or −1: error; param-s: (only) *fd*: file-descriptor (must be opened); after/with *close*(): {*fd* no longer used anymore; kernel can remove associated data-structures; indentifier *fd* can be reused later}; by default: process termination automatically closes all open files}.

12 "The Final Lecture" (imple of File-System abstractions)

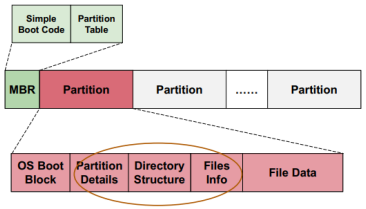
Overview: {F-S imple: {f-sys layout; disk organization}; Imple details for: {file-info; free-space management; dir structure}; file-system in action ("mini case studies")}.

More Overview. File-sys-s are stored on storage media (so, we use data in media to utilize the media). We concentrate on hard disk (in this lecture). General Disk-Structure: {1-D array of logical blocks; logical blocks := smallest accessible unit (usually 512B-4KB); logical block is mapped into disk-sector(s) (the layout of disk-sector is HARDWARE-DEPENDENT)}.



Note: partition in windows is similar to your "C:" and "D:" drives in windows. If you want, each partition may have its own OS as well.

Generic Disk Organization



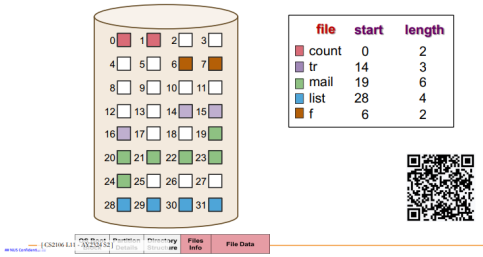
Note: depending on imple, the three middle-sections may be combined. Also, file info is minimally metadata.

Disk Organisation: {MBR/Master-Boot-Record at sector 0 ("small area in front of partition table"), with partition table; followed by 1 or more partitions (each partition can contain an independent file system)}. A FS generally contains: {OS Boot-up info; partition details: {total num of blocks; num and location of free disk blocks}; dir-structure; info of files; actual f data}.

File Imple Overview. A file can be logically viewed as a collection of logical blocks. When file size != multiple of logical blocks, last block may have internal fragmentation.

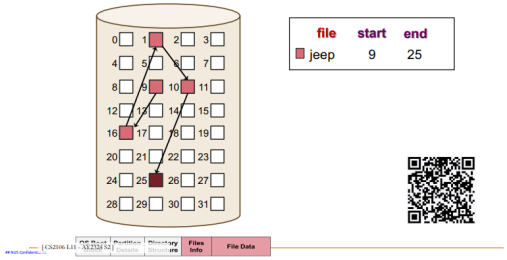
A good f-sys imple must: {keep track of the logical blocks; allow efficient access; effective utilization of disk space} ⇔ the difference of file systems is their focus on "how to allocate" file data to logical blocks.

File Block Allocation 1: Contiguous



General idea: allocate consecutive disk blocks to a file. Pros: simple to keep track, fast access; Cons: external fragmentation, file size need to be specified in advance.

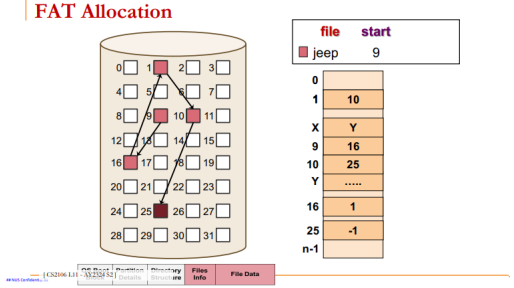
File Block Allocation 2: Linked List



General idea: {keep a linked list of disk blocks; each disk block stores: {the next disk block number (i.e. stored as and act as a pointer); actual file data}; f-info stores first and last disk block num}. Pros: solve fragmentation problem; Cons: {random access in a f is very slow (i.e. "a lot of moving around"); part of disk block is used for ptr; less reliable (what if 1 of those pointers is incorrect?)}.

Note: this linked-list can be optimized; for example, by adding additional pointers "intended for long traversals" (but still $O(n)$ aggregate-ly).

FAT Allocation

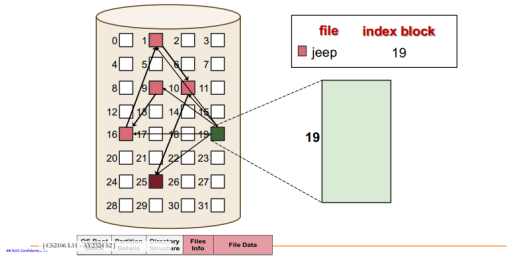


General Idea: {move all the block ptr-s into a single table (known as FAT, which stays in memory at all time); simple yet efficient}.

FAT allocation table details. FAT entry contains either: {FREE code (block is unused); block-num of next block; EOF code (i.e. NULL ptr to indicate it's the last block); BAD block (unusable block, i.e. disk error)}. FAT version 16 implies 2^{16} disk blocks; version 32 implies 2^{32} disk blocks. Example: block 3 → 5 → 8 → EOF.

Pros: faster random access, since now the linked list traversal takes place in memory; Cons: FAT keeps track of all disk blocks in a partition, {can be huge when disk is large, and so consume valuable mem space (expensive overhead)}.

Indexed Allocation



"Can we do better?" Answer is yes: allocate a memory block for "positions where (currently) open files are stored", known as *index table*. Why? Don't need every file information at every time, just need those of which are open in RAM. (from what I'm catching, every file has a (unique) index block)

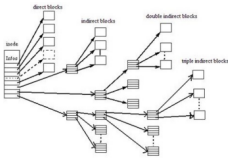
General Idea: {instead of keeping track of every f in the system, we maintain blocks for each file; each f has an index block: {array of disk block addresses; with $IndexBlock[N] == N^{th}$ Block address}}. Pros: {less(er) memory overhead, since only index block of opened file needs to be in mem; fast direct acess}; Cons: {limited max f size, since "max num of blocks" == "num of index block entries"; (existence of) index block overhead}.

Variation (schemes to allow larger file): {linked scheme: {keep a linked list of index nodes; cons: expensive since traversal cost (can be large)}; multilevel index: {similar idea as multi-level paging; can be generalized to any num of levels}; combined scheme (used in Unix, as Indexed-Node or I-Node): {combination of direct indexing and multi-level index scheme; fast access for small files (use direct index); handle large files (use multi-level)}}.

Notes. 1st scheme: if need more than 1 index block for that file (linked list of index block, still have disadvantage of $O(n)$ traversal).

I-Node Data Block Example

- Direct blocks:
  - 12 data block x 1KiB = 12 KiB
- Single Indirect block:
  - Number of entries in a disk block = 2^{10} / 4 (i.e., block address) = 256 entries
  - 256 x 1 KiB = 256 KiB
- Double Indirect block:
  - 256^2 x 1 KiB = 64 MiB
- Triple Indirect block:
  - 256^3 x 1 KiB = 16 GiB



25

I-Node Data Block Example. Every f/dir will have a I-node associated with it (uniquely-numbered I-node). Design of I-Node allows fast access to small file AND flexibility in handling huge file.

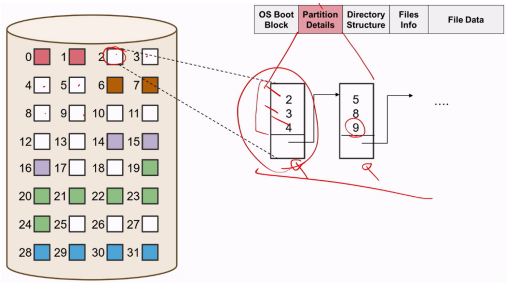
The actual example: e.g. consider 4-bytes block addr + 1KB disk block (i.e. 1 disk-block is sized 1KB). We assume "main directory of disk blocks" has 12 pointers (i.e. 12 direct blocks) + single, double, triple indirect blocks (how big of a file can be stored in a single structure of single/double/triple indir-blocks?)

Dir-blocks: 12 × 1KB = 12 KB. Single indirect block: {num of entries in a disk block = $2^{10}/4$ (i.e. 4B addresses, so total/4B addresses) which is equal to 256 entries; 256 × 1KB = 256KB}. Double indirect block: 256^2 × 1KB = 64MB. Triple indirect block: 256^3 × 1KB = 16GB.

Free Space Management. Overview (of "Partition Details" section of the slides) here. To perform file alloc: need to know which disk block is free. Free space management: {maintain free space info; allocate: {remove free disk block from free space list; needed when f is created/enlarged (i.e. appended)}; free: {add free disk block to free space list; needed when f is deleted/truncated}}.

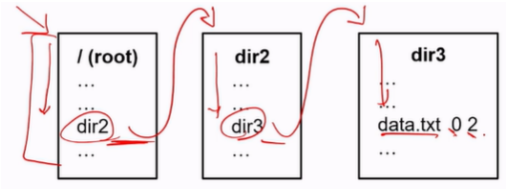
Bitmap: each disk block is represented by 1 bit (in slides: free blocks are "green-colored and 1-labeled", occupied are "red-colored and 0-labeled"). Pros: provide a good set of manipulations; Cons: this bitmap needs to be kept in memory (otherwise, hardware access is not efficient) (though, a note on the overhead: nowadays, the size of your hardware can be HUGE

How to do better? Can think of the same argument as with indexed-allocation: "do I need ALL the data/information of which memory-spaces are free (at all times)? i.e. do I need every free block which is present in my memory?" Response: just need a small set of which blocks are free, and when they are exhausted, bring up the next set of free blocks.



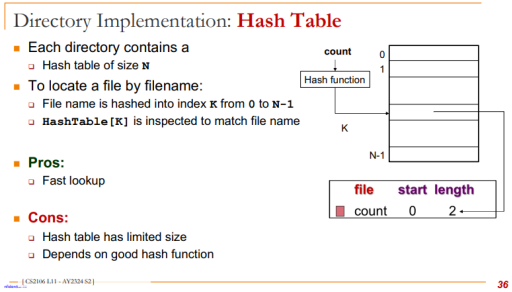
Linked List: each disk block contains {a number of free disk-block num-s, or; a ptr to the next free space disk block}. Pros: {easy to locate free block; only the 1st ptr is needed in mem (though other blocks CAN be cached for efficiency)}; Cons: high overhead (we still need to store these in our "partition detail" section; to reduce overhead: can allocate the free blocks themselves to store this "what are the next free blocks" info).

Directory Structure Overview. 2 main tasks of dir-struct: {1. keeps track of files in a directory (possibly with f metadata); 2. map f-name to the f-info}. Details: {f must be opened before use (use *open*("data.txt")); the purpose of this operation is to locate f-info using pathname + f-name ("+" as in appended)}. How does the *open* thing work (???) given a full pathname, need to recursively search the dir-s along path to arrive at the f-info "*dir2/dir3/data.txt*" (note: sub-dir usually stored as f-entry with special type, in a directory).

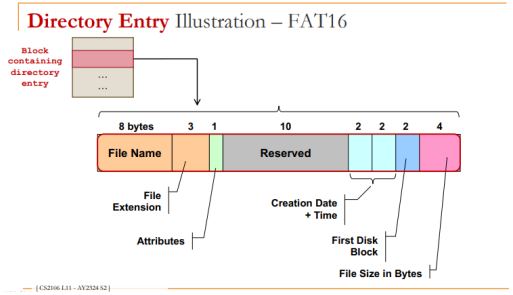


Linear List. Each entry of linear list represents file (f-name + metadata, f-info or ptr to f-info). Locate file using list (linear search; inefficient for large dir-s and/or deep tree traversal). Common sol: use cache to remember latest few searches.

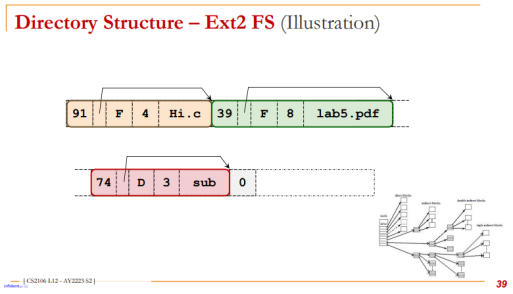
Hash Table. Each dir contains a hash table, size N. To locate f by f-name: f-name is hashed into indek K, $0 \le K \le N-1$. $HashTable[K]$ is inspected to match f-name. Pros: fast lookup; Cons: limited-size hashtable; depends on good hash fn.



File Information. Consists of f-name + metadata and disk-blocks info. 2 common approaches to store: store everything in dir entry OR store only f-name, and points to some other data structure for more info (i.e. first approach but abstracted/lazily evaluated).

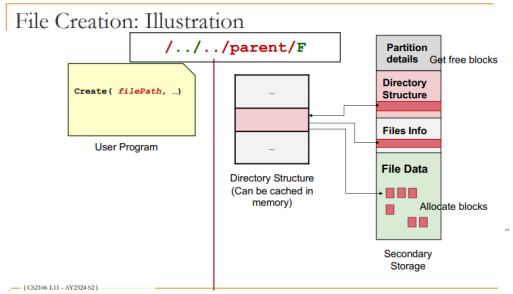


Note: reserved is if need to extend some part of the directory entry. Also, "first disk block" is the pointer to address of 1st-d-block.



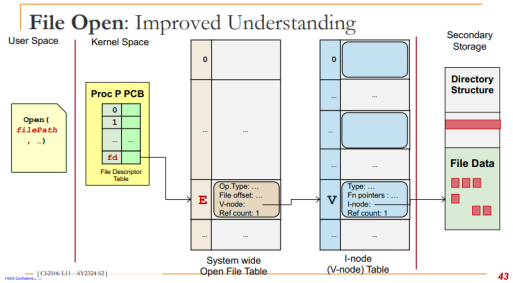
First number is I-Node num, second is offset/"ptr" to next entry, 3rd is F/dir/subdir, 4th is f-name size.

File Systems In Action.



CREATE file operation (walkthrough). To create file *.../.../parent/F*:

{Use full pathname to locate parent dir (search for filename *F* to avoid duplicates (if found, f-creation terminates with error) and search could be on cached dir structure); use free space list to find free disk block(s) (depends on alloc scheme); add entry to parent dir (with relevant f-info: f-name, disk-block info, etc}.



OPEN file operation (walkthrough). When proc *P* open file *.../.../.../F*:

{Use pathname to locate file *F* (not found → open operation terminates with error); when *F* located, its f-info loaded to new entry *E* in sys-wide-table and *V* in I-node-table (if not already present, do this for each of the two); creates an entry in *P*'s table to point to *E* (i.e. f-descriptor) AND ptr from *E*'s entry to *V* (same thing, but this is a bridge from second table to third table); return f-desc of this}. Note that the returned f-desc is used for further r/w operation(s).

Note: I-Node is (persistent) secondary storage, V-Node is I-Node entries in table. The terms "V-Node Table" and "I-Node Table" can be used interchangably. I-Node is located in disk (and always valid), V-Nodes located on kernel space (i.e. OS' memory), and only exist when a file is opened. However, (intuitively,) just one V-Node exists for every physical file that is opened.

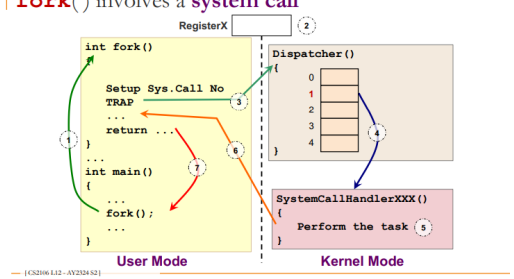
13 Big Bang!

Will just put stuff I may have forgotten here.

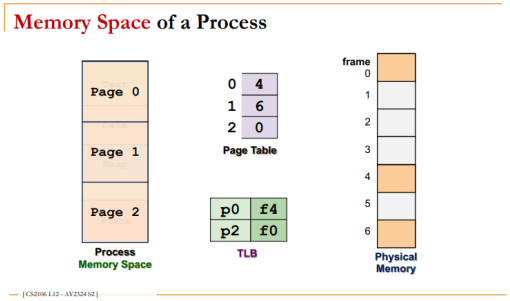
Typing "*ls*" to a shell. First, while shell is waiting for user input, it is at blocked state.

When user presses *l* on keyboard, hardware will send interrupt to interrupt handler (interrupt handler := intermediary between OS and hardware). The interrupt handler will scan interrupt vector table (say keyboard-handler number is 3) for entry 3, which points to keyboard-interrupt-handler (for this particular hardware, which is keyboard).

After pressing enter, it moves to ready state then running state.



Explanation. 1. Code forks and call library. 2. Library puts some value [that indicates that *fork*() is calling kernel mode] in some register (say, *registerX*) and syscall. 3. TRAP/switch to kernel mode + dispatcher see *registerX*'s value and look up to its table (containing routines to be executed) — this is similar to IVT which we just discused. 4. Execute *syscallHandlerXXX*. 5. Do task, with special return. 6. Return to library-code-function. 7. Library code returns to user-code/program-code.



Effect on *fork*() towards memspace. Note that mem-context is out of 3 things in PCB (hardware, mem, OS context); and this PCB is 1 entry in the (whole) process table.

Memory context of the PCB contains page-table-pointers (page table nowadays are huge), NOT contain physical frames, contains CPU registers (i.e. page-table-pointers), NOT (usually) contains entire TLB (since TLB is flushed) — but in this case, depends on implementation [if want to be efficient (i.e. not miss "as much" in earlier accesses) then it's feasible to implement].

wait(): parent context switch [so OS/CPU remembers where to continue after it runs (again)].

exec(*üsr/bin/ls*⌧): we need file-system OS service now (need to know if file exist).

File system recap. F-Data: (logical) blocks of actual contents. F-Info: metadata corresponding to file (helps access files in F-Data). Dir-Struct: all details corresponding to files and subdirectories stored in a particular dir. Partition Details: info about free and allocated blocks (etc).

Traversal for some "*ls*" file as follows: