CH4 — The process

- The idea is to virtualize the CPU. The OS does this by combining 2 things – **mechanisms** and **policies.**
- Mechanisms are low level methods or protocols that the OS calls to perform some functionality.
- Policies are the high-level intelligence that enables the OS to make a decision – for example scheduling policy.
- The process constitutes of a machine state which comprises of three things – **memory**, **registers**, and **I/O information**. **Memory** is where the process would store information and retrieve instructions from. **Registers** are what are essentially updated when a program executes an instruction. **Program counter**/ instruction pointer – which instruction to execute. **Stack pointer** + **frame pointer** are used to manage the stack with associated function parameters, local variables, and return addresses
- Process API – Consists of several API – **Create**, **Destroy**, **Wait**, Miscellaneous Control (**Suspend** and **resume**), **Status**.
- **Process creation in some detail**: **(1)** The OS first needs to load the code and the required Static data into memory. This can be done either *eagerly* (all at once) or *lazily* (Only, the code and data needed now). Since, this information lies on the disk, the OS reads those bytes from the disk and places them in memory.**(2)** Once done, the OS initializes the Stack (for function parameters, local variables and return addresses) and the Heap (for explicitly requested dynamically-allocated data for DS like Linked List, hash tables, trees etc). Allocation –> malloc(), free –>. free() **(3)** The OS would also perform some initialization tasks related to I/O for standard Input, Output and Error.**(4)** Finally, the OS would now start the program execution by transferring the control of the CPU to the main() function thus beginning the execution of the program.
- Process states – **Running** (Executing instructions), **Ready** (Ready to commence execution but waiting for OS to schedule it), **Blocked** (Performing some I/O function), **Initial** (When created), **Final** (exited but not yet cleaned up). The OS should be able to **track** in someway the processes that blocked, or when an I/O completes (**Process List** – *Data Structure*). There are important information (mem-size, bottom of kernel stack, process id, parent process, process state etc) about a process that the OS keeps a track of. The **register context** will hold the contents of the process' register while it is on hold. When restored, the OS can resume running the process. The process can also be in an **Initial state** (when created) and **Final state** (*to allow the parent to examine the return code from the child process; allowing the parent to make a final call [wait()] to clean up relevant DS*)
- **Process Control Block (PCB)** is an important OS DS that keeps information of the processes that are present in the OS, also known as the Process Descriptor – Process identification data, Process state data Process control data

CH 7 Scheduling

- Assumptions being made – **(A)** Each jobs runs for same amount of time, **(B)** Each job arrive at the same time, **(C)** Each job runs to its completion, **(D)** No I/O, **(E)** Run-Time of each job is known.
- Scheduling Metrics – ***Turnaround Time***(***TT***)[*Performance Metric*]: it is the time that the job completed execution - job entered the system; that is, $\mathbf{T_{turnaround} = T_{completion} - T_{arrival}}$. The other metric is *Fairness*.
- First In First Out/ First Come First Serve Scheduling (**FIFO**) – In this scheduling, keeping the above assumptions intact, we see that it would work relatively well. However, it would encounter problems, when the Assumption **(A)** is no longer being considered. Meaning, that if the first job that entered is VERY long, it would lead to the **CONVOY EFFECT** in which relatively-short potential consumers of resource get queued behind a heavyweight resource consumer.
- Shortest Job First (**SJF**) – To solve the problem of convoy effect presented in the above point, we can use the SJF algorithm. In this, **given the Assumption (B)** SJF would be an optimal algorithm. However, on relaxing assumption **(B)**, we see that SJF may no longer perform well. Say, if a Long Job (100 seconds) arrives first, and then relatively short jobs arrive after 10 seconds. Because of Assumption **(C)** the shorter jobs cannot be scheduled.
- Shortest Time to Completion First (**STCF**) – To solve the above problem, we should be able to pre-empt the longer job, midway so that the shorter job can be executed. For this some low-level machinery would be required that would help the OS to take control and carry on with the context switch. This algorithm can also be called the **Preemptive Shortest Job First**. So, in the above scenario, when the longer job is scheduled, it would run for 10 seconds and then when relatively-shorter job arrive; the OS would switch to the shorter jobs, which when completed could then lead to the resuming of the older and longer job that was preempted. Given the Assumptions **(D)** and **(E) STCF** would be a great policy.
- Another Metric – ***Response Time*** (***RT***): It is defined as the Time the job was first run - the time the job arrived in the system, that is, $\mathbf{T_{response} = T_{firstrun} - T_{arrival}}$.
- Round Robin (**RR**) – In this policy each job is run for a **Scheduling Quantum**. This SQ is a multiple of the time interrupt, that is, if TI is 10; SQ will be 10, 20, 30 etc. Hence, RR will help the system to become more response, as the jobs would be scheduled sooner. Therefore the Goal with a system running RR is to have scheduling quantum **large enough** so as to amortize the *costs associated with context switch*, but also **low enough** so that the system does not become unresponsive. Because frequent context switching would lead to performance degradation. Also, RR would be the worst policy if Turnaround Time is the metric under consideration. Hence, so far two types of scheduling policies have been introduced; The first – SJF, STCF which prioritize **TT**, and RR which is a **fair** policy prioritizing **RT**.
- Relaxing assumption **(D)**, now with I/O into the picture, a job would give up the CPU, hence when another scheduling policy is taken into consideration such as **STCF**, another job can be scheduled in between the first job's I/O activity.
- Assumption **(E)** is relaxed in the next chapter with **MLFQ**.
- For jobs with same length SJF delivers same turnaround time as FIFO
- When job length = Scheduling quantum SJF delivers the same response time as RR
- There is a linear relationship between Job length and Response time for SJF
- As job lengths increase average response time goes up. If the jobs are sorted in increasing job size order, then the n-th job's response time is equal to the sum of the execution time of all the previous n-1 jobs plus that of the current job. Increasing the size of any job thus increases the response time for it and all larger jobs.

CH 16 – Segmentation

- When a program is divided into three continuous segments for code, stack, and heap, a lot of memory is wasted. Hence, the previous method of just using one base and bound register per program would prove to be wasteful, hence now we describe a process of having **per-segment base and bound register** which would allow us to keep each of the segments in **different physical memory** location, and still make the program thing that a continuous private memory is allocated to it.
- Therefore for the code segment, the virtual address would be added to the base register to get the physical address; however, for the heap, we would first calculate the offset that would be equal to the **virtual address - the offset register + base register value**. If a reference to an illegal address is made, it would result in a **segment violation**.
- To determine which segments we are referring to, **EXPLICIT WAY**: we can simply divide the 14 bits used to define a virtual address into two parts. The first two bits can help us whether we are referring to the Code Segment (**00**), Heap Segment (**01**) or Stack Segment (**11**). The remaining 12 bits would then help us get the offset. For example the virtual address 4200 in binary is represented as follows
  - **01** *0 0 0 0 0 1 1 0 1 0 0 0*
  - **01** –> Represents that the segment is Heap
  - *0 0 0 0 0 1 1 0 1 0 0 0* –> Gives the offset, that is, 104.
  - Also, in some cases, the code segment is stored along with the heap, hence in some systems only 1 bit is used to identify the segment.
- **IMPLICIT WAY**: In this method, the segments are identified implicitly based on where the address is generated. **Code Segment**: Address was generated from the program counter. **Stack Segment**: Address is based off of the stack or base pointer. **Heap Segment**: Any other address generated.
- However, Segments grow negatively, hence we need an additional bit specifies whether a segment grows positively or not. Once we get that, for segment we simply would have to subtract the offset from the base segment or add the negative segment to the base register value to get the physical memory location of the segment.
- Also there is an availability of a protection bit in case of memory sharing. In addition to checking whether the address is legal or not there would be an additional check whether the memory is now shareable or not.
- The issues that are raised with segmentation is that the OS get to get more and more fragmented, due to the variable size of each of the segment being allocated. This results in an issue called **EXTERNAL FRAGMENTATION**, which simply means that even though enough (total) memory is available to be allocated to a new process, the OS cannot do it because it is not available in a continue segment. To solve this we can perform something that is called **COMPACTION**, which simply accumulates the free memory available in a single continuous block. Another way to proceed with this issue is to have a free-memory list, that the OS can use to keep track of all the free memory and an algorithm that would manage the free memory.
- Unfortunately, though no matter how smart the algorithm is, external fragmentation will still exist; but will simply work towards minimizing it.

CH5 — The processAPI

- The **Fork()** system call– It provides a way for the OS to create a new process which is the **exact copy** of the calling process, which makes the OS things there are two programs which are about to return from fork(). The child process however, doesn't start executing at main(). Also, the child is **not the exact copy**, meaning that even though it has its own address space, registers, PC, etc, value it returns to the caller of fork is different. The **parent** receives the **PID of the child**, the child receives a return code of **0**. And the output for a parent calling a child function would not be deterministic, meaning that either of them will run first. – Scheduling issue.
- **Wait()** System call –This will make the parent process wait for the child process to complete execution before commencing its execution. This in turn makes the output **deterministic**. Because if the Child is scheduled first no issues; however, if the parent is scheduled first, it would then call the *wait* function to allow for the child to be executed first.
- **Exec()** System call – This function will result in the execution n of another program other than the calling program. So, when now, the child is created and a call to an exec() function is made, it would then execute the program being called in the exec function, **however, it would not continue the execution of the same program**, meaning that any line of code written just after exec() would not be executed; since the heap and stack segments are not re-initialized. And it **overwrites** its current code segment.
- Also note that this **does not create a new process,** it transforms the currently executing(child) process into a different running program. **A successful call to exec() never returns**.
- Meaning, now after the exec() function call, **directly the code associated with the remainder of the parent will be executed**.
- There are other system calls as well such as *Kill* which would kill the process when the PID is specified. However, this brings to the notice about which processes will be able to call the kill function since it would be a critical function. Hence we have the notion of the User and the SuperUser which would be given the responsibility of managing such tasks.

CH 8 Scheduling — MLFQ

- The idea with MLFQ is to make scheduling decision without future knowledge as discussed in Assumption **(E)**. This would require MLFQ to focus on two things *optimize **TT** and minimize **RT***.
- MLFQ uses several queues differentiated based on the **priorities** of each of the queues. Each ready job would be running on one Queue. **Rule 1**: Higher Priority Jobs are preferred. **Rule 2**: Jobs with same priorities are run in a **RR** fashion. Hence, MLFQ uses history to predict the future.
- MLFQ varies the priorities based on observed behavior. **(1)** If a job relinquishes the CPU frequently, it would get a *higher priority*, as opposed to **(2)** jobs that uses CPU for Long time i.e, *lower priority*.
- Changing priorities is one of the challenges with MLFQ. **Rule 3**: A *new process* would have the highest priority. **Rule 4a**: If a job uses the entire time slice, its priority would be reduced. **Rule 4b**: if a job relinquishes CPU before time slice expires, its priority would be unchanged.
- Examples – **(1)** One Long running job *(200ms)* [**Q1**]; time slice = *10ms*; its priority would be lowered after 10ms [**Q2**] and then again after *10ms* [**Q3**]. (2) Short Job (*20 ms*) @ *100ms* [**Q1**]; after *10ms* reduced to **Q2** when it completes execution. Because MLFQ **doesn't know** whether a job is short/long it **assumes** it is short. If short, it would complete execution soon; else it will move down resulting in a long-running batch-like process. **This is how MLFQ approximates SJF**. **(3)** When there is a job which relinquishes CPU bc of I/O *before* time slice expires, priority would not be changed.
- Problems with this approach – **Starvation**: Too many short jobs overwhelm the CPU, **Gaming the scheduler**: a program deliberately gives up CPU before the time slice expires to avoid priority reduction, **Change of Behavior**, the process initially used a lot of CPU, but no longer does, yet it is lowered in priority.
- Using priority boost to solve some problems – **Starvation** and **Change of Behavior**. **Rule 5**: After some time period *S* increase the priority of all jobs to the topmost queue. If *S* is *too high*, then the long-running jobs would *starve*, if it is *too low*, then *interactive jobs wouldn't get proportional share.*
- Preventing **Gaming of scheduler**: **Rule 4** (replace **Rule 4a** & **Rule 4b**): The OS would keep a track of the time slice being used by a process. Hence, once a job uses up its time allotment at a given level; *regardless* of how many times it has given up the CPU, its priority is reduced.

CH 19 – Paging Faster Translation

- With paging, the major challenge that needs to be overcome is its speed. This is overcome with the help of hardware help, down as the Translation Look-aside Buffer (**TLB**). This TLB would be holding the information that would be needed to translate a virtual address into a physical address. If the TLB has the information, it would be known as a **TLB Hit**, and the translation would take place, if it does not have the required information, it would be called a **TLB Miss**. In this case there would be the cost associated with paging that would be incurred. Once the Page Table Entry is calculated, it would be checked whether it is valid, or if it is accessible, if so TLB would cache the information and then the virtual addr would be translated into a physical address.
- **Spacial Locality** – It is taken into consideration when most of the entities are very close to each other, and while accessing one of them, increases the probability of a TLB hit for the other memory locations with the entities near them. **Temporal Locality** – It is taken into consideration, when a block of memory recently accessed is accessed again, which increases the probability of achieving all Hits, due to the relative indifference in time, due to TLB caching. Also, Page size is inversely proportional to TLB Misses
- The management of TLB misses was in olden days done by the Hardware, but now with modern systems, it is done entirely by the OS. In the case of a TLB miss, OS would simply raise an exception, which would then handled by the trap handler, however, there is a key difference. In case with TLB misses, the return from trap function would require the execution of the instruction that caused the trap in the first place, and not the execution of the next instruction as would be the normal case. Additionally, OS needs to ensure that it does not cause an infinite chain of TLB misses, which could be solved by having the TLB miss handlers in the physical memory.
- TLB issues with **context switches**. It is imminent that two different processes share the same virtual address, pointing to different physical frame. This would lead the TLB to get **confused** when context switch takes place. To solve this problem, one of the ways to go ahead would be to make the **TLB flush** every time on a context switch, **but** that would cause a lot of **overhead** because it would lead to TLB misses as it touches the code and the data pages. To solve this, instead of flushing, we can use a **Address space identifier** (*ASID*) which is akin to the process identifier (*PID*), with less bits (8 bits) to identify to which process a particular virtual address maps to. There could also be processes, that refer ti the same physical frame number, as in cases where different programs share the code segment, which is common and OK.

CH6 — Limited Direct Execution

- The challenge with virtualization is that we need to maintain high Performance and at the same time enable the OS to have control.
- **Limited Direct Execution** –Simple way is to directly run the process on the CPU.
- *Problem 1* **Restricted Operations** – How to restrict execution operation by a process. Two processor modes – **User mode** and **Kernel Mode**. In the user mode, the process cannot execute privileged operations. Whereas in kernel mode it can. To allow for this the hardware allows what is called **system calls** to be made, to allow for execution of privileged operations. To execute a system call, the program should execute a **trap** instruction which jumps into the kernel mode to raise the privilege level to kernel mode. When the execution is completed, the OS would call a **return-from-trap** instruction to return into the calling user program – simultaneously reducing the privilege level to user mode.
- The hardware, needs push certain information pertaining to the calling user program into a **per-process kernel stack**, which would store important information such as *Program counters, flags and other registers*, which would help the hardware to reload the program execution once the system call is executed.
- However, how does the trap instruction know which system call to execute. **Two Phases in the Limited Direct Execution protocol**:
- To allow for this the OS on **(1) boot-time initializes a trap table** which associates each of the system call to a particular system call number. **(2) The kernel sets up a few things** (allocating memory, node in process list) **before using a return to trap to start execution**. The OS would inform the hardware about these **trap handlers** which would be responsible for handling trap instruction calls from the user program. This information would be remembered by the hardware until the next boot happens.
- Hence the User program just needs to remember the system call numbers, which would be placed into the trap instruction. The hardware would then assert the system call number, and then refer the trap table to know the location of the system call to be executed, and executes the system call. This level of indirection provides for protection from malicious execution of code. Switching between processes.
- *Problem 2* **Context Switch** When a process is running on CPU, OS is not. Hence, how can the OS regain control of the CPU. **Cooperative approach** – in this the OS waits for the process to make a system call. Essentially it **trusts** the process. However, if the program/process deliberately/ accidentally is stuck in an infinite loop, the OS will not ever regain control. **Non-cooperative approach** – In this, there is the introduction of a time interrupt whose value (in ms) would be initialized in the interrupt handler. OS informs the hardware about this and during the boot sequence the OS would start the timer.
  - Now it is the responsibility of the hardware to ensure that enough process state information is stored in the corresponding general purpose registers, PC, kernel stack pointer, and restore the PC and said registers of the soon to be executed process onto its kernel stack (**context switch**) when the interrupt goes off, so that the process currently being interrupted can be resumed later/ now whenever it is scheduled.
- There are two types of register saves. **First** is when the timer interrupt occurs – in this case the user register are implicitly stored by the hardware using the kernel stack of that process. The **second** is when the OS decides to switch from A to B; in this case the kernel registers are explicitly saved by the OS, but this time into memory in the process structure of the process.

CH 13 – Abstraction: Address Space

- In virtualization of memory, each of the process would have their own virtual address space which would be mapped to the physical memory in some form. Each address space for each of the process needs to have enough information to allow for easy store and restore of context during the context switch. Hence, each address space, consists of at least three key elements (though there are more) – **Code** (instructions), **Stack** (local variables, parameters and return values; *Grows Upwards*) and **Heap** (dynamically allocated memory from user side; *Grows Downwards*)
- Goals with virtualization – **Transparency**: Meaning that the program should be unaware of the fact that the memory it using is virtualized. In fact it assumes that the memory it accesses is its own private memory. **Efficiency**: Both in terms of *time* and in *space*; for which the OS needs help from the H/W. Finally, the third goal is **Protection**: meaning that the OS should protect the memory of one process to be accessed by another process, as well as the OS itself thereby enabling **isolation**.

CH 15 — Address Translation

- The address translation would take place with the help of both Hardware and OS. Hardware would provide the low-level mechanisms to help with the translation, where as OS would help with the memory management.
- The challenge is to virtualize the memory in such a way that even though the process sets the start of the code segment at position 0, the actual address in the physical memory would be way different, and the OS needs to make sure that translation takes place with ease such that the process feels that it has its own private virtual memory but, it would still use the physical memory to address the core segments – code, stack and heap.
- **Dynamic Hardware Relocation**: In this method, there are two physical registers (**memory management unit**) residing on the (one pair per CPU) CPU – **Base** & **Bound** that would help with the translation of the virtual address into physical address @ *runtime*. The Base would be the address, from which the program's code segment starts, and Bounds would help in protection of the memory from going overboard than what is allocated to the program. Hence we get; ; where base would be the value in the base register. The bounds register would be used to just verify whether the memory address to be accessed in a particular instruction is within the range of a particular program.
- Operating System Issues. **First**: When the process is first created, OS must make sure it allocates a position on the memory for the process. To do this, OS initializes a free list, keeping a track of all the physical memory slots that are free and available to be used by a process. **Second**: OS must clean up the memory once the process is terminated and add the cleared up memory back to the free list to enable it to be allocated to other processes. **Third**: During Context switches, it is the responsibility of the OS to store and restore the base and bound registers for each of the processes, because there is only one pair of base and bound registers available per CPU. Even when the a process is stopped, it can be descheduled by the OS and moved to another location on the physical memory, while storing and restoring the base and the bound register on the CPU. **Fourth**: OS should provide exception handlers to deal with exceptions that take place during runtime associated with the memory. This could be faulty use of privileged operation by the user in the user mode, or could even be accessing a memory out of the bounds from the particular processes' limits.
- Physical address = virtual address + base **(out of bounds Fault is possible)**

CH 21 – Beyond physical memory: Mechanism

- To allow for a larger virtual memory than what the physical memory allows, we would have to use the disk for the storage of the pages to be swapped in and out called **swap space**. To do this, some disk place would have to be received for such swapping to happen, and the OS also needs to remember this disk address for a given page.
- So there are two possibilities with the use of TLB – **TLB Hit** or **Miss**, in the case with miss, the OS would have to look into the swap space of the disk to determine whether or not the page is present in the disk. To enable this, we need additional machinery to support this, specifically a present bit that would tell the TLB, whether the page that it is currently searching for is in the physical memory or in the disk somewhere. The act of accessing a page that is not in the physical memory us commonly referred to as **page fault**.
- So when a page fault is encountered, the OS would then look into the PTE to find the address in the disk and then use this to issue and I/O to fetch the page back into memory. While this happens, the page would be in a *blocked* state, and the OS can run any other process. Once the I/O is complete, the OS would update the PFN field of the PTE to record the in-memory address of the page. When the TLB tries to access the page, it would get a TLB miss, and then it would be loaded in the TLB, which when the return from trap function happens would result in a Hit. If however the memory or the swap space is full, a page replacement policy would be used to load the desired page in memory by swapping the not-required pages.
- There are three scenarios in which TLB miss might occur. **First** is when the page is both valid and present, TLB miss handler would simply grab the PFN from the PTE. The **second** case is when a page fault occurs, that is, the page is not present in the physical memory. The **third** case is when the page being accessed in invalid.
- To keep a small amount of memory free, the OS would keep some kind of ***high watermark*** and ***low watermark***, which would be responsible for keeping the memory in check. If the memory available Is lower than the low watermark, then the eviction of pages start, until the free memory reaches the high watermark level. All this is done by a background thread called the swap daemon.

CH 22 – Beyond physical memory: Policies

- The main focus would be to minimize the number of cache misses. To calculate the Average Mean Access Time (**AMAT**) is (1) , where T_M is the cost of accessing memory, T_D is cost of accessing disk, and P_miss is the probability of missing a cache. The value of T_M<<T_D , hence it is imperative to reduce the number of cache misses.
- Optimal cache replacement policy. In this case, the page that would be accessed the farthest would be replaced. Also, if the cache size is , then there would be number of misses, credits to **cold start** or **compulsory misses**.
- **FIFO Policy**: In this the page that was accessed the first would be evicted first. However, there is special case with *Belady's Anomaly*, in which increasing the cache size increases the miss rate.
- **Random Policy**: In this policy, the pages to be evicted are randomly chosen. In some cases, Random does better than the FIFO policy and as good as the Optimal policy, but it is entirely dependent on the luck of Draw.
- **Least Recently Used**: LRU and LFU are based on the *Principle of Locality*. LRU keeps in mind the the history of the pages being used, and would then evict the page that has been Least Recently Used. Where as LFU, would keep in mind the past frequency of usage of the pages and would then replace the page that has been Least Frequently Used.

(1) $AMAT = T_M + (P_{miss} \cdot T_D)$

CH 26 — Concurrency: An Introduction

- **Threads** allow multiple point of execution of a program. Thread is the same as a separate process, except for the fact that it would share the *same memory address*. The thread would also have a program counter and a private set of register it uses for computation. With processes we have a process control block to save the state of the process, with threads we have the **Thread Control Block (TCB)**. Finally each thread would have its ***own stack***, meaning that if there are two threads for a program, each of them would have their own stack.
- There are two basic uses of threads, **First** – Parallelism. This allows us to speed up programs and their execution. **Second** – To avoid blocking of a process during I/O. In such a case, one of the threads could be waiting for the I/O to complete, whereas other threads can continue execution and utilization of CPU in some manner.
- Thread creation, is the process in which the main thread currently executing, creates more threads, and then passes over the execution to these threads. Once the threads are created, they are scheduled by the OS using some algorithm. Like, processes, these threads can be scheduled in any order, and hence the final result would be **indeterministic**. Additionally, with threads, there arises one more **challenge** – **shared data**. When separate threads, interact with shared data, there is a possibility of the rise of a **race condition**, that is, there are multiple threads that have entered the critical section at roughly the same time; and they attempt to update the shared data structure. Simply take the counter example. In this example, since multiple threads are accessing the same shared data (*counter*) this code is called the **critical section**. Critical section could also be a code which accesses the same resource. What we really need is, a property called **mutual exclusion** that would ensure that if one thread is executing in the CS, no other thread would.

CH 27 — Interlude: Thread API

- **Thread creation** – The function used is pthread_create and it takes 4 parameters. The first parameter is the pointer to a structure of type pthread_t which is used to interact with the thread. The next parameter is attar which is used to specify any attributes that this thread might have like stack size or scheduling priority, which would have to be initialized using pthread_attr_init() . The next parameter is the function that this thread would start running in; also called the **function pointer**. Finally we have the parameter, which simply takes the arguments of the function that the thread would be running in
- **Waiting for a thread** – The function used is pthread_join . This would wait for a thread to complete. Takes two arguments, pthread_t and the second argument is a pointer to the return value you expect to get back. However, there are three points to note. **First**: We do not always have to do all the packing and unpacking of arguments, that is, we can pass NULL, if thread takes no arguments. **Second**: If we are just passing a single value, we do not have to package it up as an argument. **Third**: ***Never*** return a pointer which refers to something allocated on the thread's call stack.
- **Initialize the locks and condition variables**. Failure to do so will lead to code that sometimes works and sometimes works in very strange ways.
- **Each thread has its own Stack**, hence all the locally allocated variables are private to the thread and no other thread can easily access it. To share the data, values must be present on the heap, or otherwise globally accessible.

CH 28 — Locks

- Threads – Threads share the code and heap segments, but each thread has its own stack. Process is the resource allocation unit, and thread is the scheduling unit
- Lock -> Variable -> holds the state of lock – available or acquired (by one lock)
- Store Other information - which thread holds lock, queue for ordering lock acquisition, hidden from the user of lock
- Lock() and Unlock()
- Calling a lock, enables a thread (now the owner) to acquire the lock if available thus entering CS. If another thread calls lock() it will not return, preventing threads to enter the CS.
- Once, the owner calls unlock(), the lock is released. If no thread is waiting to acquire the lock, it would simply be turned to free; else the threads **eventually** notice or are **informed about** this change of lock's state, and enter the CS.
- Locks provide minimal control over scheduling to programmers. Threads are entities created by the programmer but scheduled by the OS.
- Pthreads Declare a var and you should have it init to call the function
- If diff lock var, then these are not mutually exclusive, coz they have to be on the same critical section. Benefits – Parallelism
- Building a Lock --H/W Requirement + OS support
- Evaluating Locks
- Metrics -> is **mutually exclusive**? // **Fairness** // **performance**
- Controlling interrupts (before the CS disable and then enable after the CS)This would be just for that thread for the process requesting it
- Problem – **Trust** to not abuse the functionality. Malicious program to **monopolize** use the resource. The program is buggy, if the process is going on in a **loop *forever*** :P. Does not work on the multicore machine. Also, this is very time consuming

CH 28 — Locks

- **Using Loads/Stores** Flag variable – use a flag variable to identify whether a thread is in the CS, while entering set it to 1 (Lock) and while leaving (unlock) the CS, clear the flag. If another thread wants to enter the CS, it checks the flag, if SET, it simply spin-waits. Two problems with this – correctness, performance **Correctness –** concurrent programming first thread sets to 1, interrupted, second thread sets the flag to 1. Both now interleaving, and in CS. Performance – Spin waiting is bad.
- **Test and Set** – atomic exchange – uses hardware
- Get the value stored at old memory location, store the new value in the old men locatn, and return the old value
- The key, of course, is that this sequence of operations is performed **atomically**
- Needs a preemptive scheduler – to run different thread from time to time as a thread spinning on a CPU will never relinquish it.
- Evaluation of metrics
- **Mutual Exclusion–** Yes, this method ensures that only one thread is in the CS of the program at any point.
- **Fairness**This method is not fair, since it does not guarantee any fairness in which threads are being scheduled.
- **Performance Single Processor** The method does not fair well, since when one of the threads in the CS is preempted, the other threads would simply be spinning in the while loop, until the interrupt is called in. Waste of resources **Multi-Processor** The method fairs well, since a thread running on CPU1, when preempted in the CS, would still allow other threads to be scheduled on other processor or allow the same thread to be scheduled to complete the CS.
- **Compare and Swap** Similar to Test&Set. Takes 3 params, old *pointer, expected value, and new value. Old pointer is simply the current value of the lock->Flag, expected is 0, new is 1. If the return value from the function is 1, then keep spinning, else allow the thread to acquire the lock. "It is more powerful simply because it has the extra checking."

CH 28 — Locks

- **Load-Linked and Store Conditional** There are two separate functions load-linked [LL] and store conditional [SC]. LL simply returns the value stored in the flag. Whether it is 0 or 1. Now Store conditional is where the actually acquiring of the lock happens. First check if the value in the load linked has been updated so far, if so fail the acquisition of lock as two threads are competing for the lock, and try again. If not, successful lock acquisition by the thread. The failure to acquire lock might happen when two threads have executed LL and checked that the lock isn't acquired, but while executing SC it fails, leading them to try again, until [while(1) would encapsulate the LL and SC function] only 1 of them has acquired the lock.
- **Fetch and Add –** Basic operations involved are using a ticket and turn variable to keep track of the currently running thread, and the next in line thread to be executed. It does so by 'fetching' the currently passed variable (turn/ticket) and incrementing it by 1.
- Now in the actual code of the locking function (after initializing the values of ticket and turn to 0), a thread gets it 'ticket' (myturn) for execution, [set to 1, for the first thread]. Now it checks if the myturn != turn, meaning that whether the current thread should be scheduled or not. If not, keep spinning while it is the threads turn, else, allow the thread to acquire the lock.
- Now during the unlock, when the thread releases the lock, it increments the 'turn' variable by 1, allowing the next thread in queue to be considered for acquiring the lock. It ensures progress for all threads. Thus no starvation. Possibly a fair method?
- **Too much spinning.**
- Avoid Spinning – need H/w Plus OS support
- Simple Solution Just yield, baby :P – Give up the CPU instead of spinning. Sleeping instead of spinning
- What will happened if the park and m->guard = 0 is interchanged?
  - The guard will never be set to 0
- When a thread is woken up from park(); it does not hold the guard. Thus the lock is passed directly from the thread releasing the lock to the next thread acquiring it.

CH 29 — Lock-based Concurrent Data Structures

- Adding locks to data structure to yield high performance, and enabling several threads to access the structure at once; **concurrently**
- **Concurrent Counters**
- Problems with counter without locks, is performance
- Simple but not scalable
- The problem with the approach wherein, we lock and unlock before initializing, incrementing, decrementing, and getting the values is that with more number of threads, **the time taken to execute is very high.**
- Scalable Counting
- In this example, we have *local counters* and equal number of local locks. The number of local counters would be equivalent to the number of CPU. There would be one global counter, and one global lock.
- Whenever, a counter needs to be updated, the local counter is updated by using the local lock. –> results in **scalability**.
- A threshold value S would be chosen, that is when the value from the local counter would be updated to the global counter. **Lower** the threshold value, more it would behave as a **non scalable** counter (low performance), **higher** the value of threshold, **better would be the performance**, which could possibly even push the global counter value.
- Concurrent Linked List
- Scalable solution is to use locks for each node rather than the whole program. If we use one thread for all the nodes, the program wouldn't scale well. Hence locks for each node. Also known as **Hand-over locking** or **Lock Coupling**. This would result in higher concurrence but is practically more prohibitive.
- **More Concurrency isn't necessarily faster.(if design adds overhead)**
- Concurrent Queues
- Add a dummy node to separate the computation of **head** and **tail** nodes. The goal is to allow for concurrency for enqueue and dequeue operations.
- Concurrent Hash-tables
- Built with the concurrent list discussed above, with a lock per hash bucket (each represented by a list).
- Avoid pre-mature optimization – **Knuth's law**

CH 30 — Condition Variables

- How to make a thread check for a condition, before continuing its execution.
- Condition variables is an explicit queue, to which the threads add themselves to, while the threads are waiting for a particular condition to come true.
- Definitions and routines –The CV has two operations associated with it – wait() and signal(). **Wait**() – thread wishes to put itself to sleep. Takes mutex as parameter and assumes mutex is locked. The responsibility of wait is to release the lock and put the calling thread to sleep (atomically) **Signal**() – thread wants to wake up a sleeping thread. A state variable is necessary as that would record the value the threads are interested in knowing. (some **identification** is needed to know which thread to wakeup). Also it is important for a **thread to hold a lock** in order to signal and wait, because a thread could change the state variable leading to parent being sleeping forever.
- **Bounded Buffer problem –** One or more producer threads, and one or more consumer threads producer produce data – put them in buffer. Consumer grab the item from the buffer and consume them
- A problem with this approach using if conditions is that when we have multiple consumers and one producer.
- What would happen when a producer produces a value, puts it in the buffer for the consumer, but the consumer 1, after initializing sleeps, and another consumer 2 sneaks in.
- When returning from the producer, the consumer 2 would be scheduled – which btw consumes the value produced by the producer.
- However, we need to still keep in mind, that there is still a consumer which is sleeping. When it wakes up to consume to previous value, the buffer would be empty. *oops…*
- A simple rule to remember is to **always use while loops** with condition variables.
- Even if the while loop is being used, there would be another problem in a similar case as above.
- Lets say, two consumer start out, and both are set to sleep one after the other.
- A producer then come in to produce and stores value in the buffer for the consumer to consume. And before it can create another value to store in the buffer, it sleeps because – well the buffer is full.
- When the consumer 1 tries to consume to value it successfully does so. However once completed; there would be two threads that it can schedule. BUT now is the problem… which one to schedule. Ideally the producer should be scheduled, but what if the consumer is scheduled. There would be no value in the buffer to consumer by the consumer. *Oops…*
- **So an important point is to have signaling – which is more directed**.
- A consumer should not wake up other consumers… only producers, and vice versa. Solution is to use two condition variables instead of 1.
- This would ensure proper signaling to the appropriate type of thread that needs to be scheduled.
- The next step would be increase the buffer size, and to make this happen we would have the producer to not sleep until all buffers are currently filled, moreover a consumer would sleep if all buffers are currently empty.
- **Covering Conditions** – In this there is a broadcast signal function that would wake up all the threads that are waiting up on a particular resource. Because, there might arise a case, wherein the normal signaling does not wake up a thread that can be scheduled immediately but wakes ups thread that cannot be scheduled at all. Take for example if a thread_A needs 100 bytes whereas thread_B needs only 10 bytes, and a free() call, frees up 50 bytes, but the thread that was scheduled to wakeup by the signal call was thread_A and not thread_B. In this case, a broadcast signal would wake up all the threads, allocating the resource to the required thread.

CH 32 — Common Concurrency Problems

- Two types of bugs – Non-Deadlock and Deadlock. Non-Deadlock bugs have two major types – Atomicity Violation and Order Violation.
- **Atomicity Violation** – In this type of bug, by definition, the desired serializability among multiple memory accesses is violated, that is, a code region is intended to be atomic, but **atomicity is not enforced**. This problem, or bug, can be solved by putting *Locks* around such code regions.
- **Order Violation** – In this type of bug, there is an inherent order in which the code needs to be executed, but **no order is enforced**. This bug can be easily solved by using *Condition Variables* that would enforce the execution of a particular code region, thereby establishing order of execution. Note, that the CV is to be placed within locks.
- Deadlock occurs when two threads are waiting for locks that are currently held by the other thread and are waiting for it to be released. Few prominent reasons for this happening are that, large code bases, often result in complex dependencies which may lead to a deadlock. Another reason, is due to encapsulation, that is, the implementation details are hidden which does not mesh well with locking.
- Four conditions that are needed for deadlock to occur – **Mutual Exclusion**: Threads claim exclusive control of resources they require. **Hold-and-Wait**: In this the threads hold on to resources and are also waiting for additional resources. **No Preemption**: In this, the resources cannot be forcibly removed from threads that are holding them. **Circular Wait**: In this there exists a circular chain of threads such that each thread holds one or more resource that is currently required by some other thread in chain.
- Prevention –
- **Circular Wait**: To avoid circular wait, enforce order in acquisition of locks. Two ways: **Total ordering** in which all the locks in the program have a predefined order in which they are acquired by threads, However, that is not always possible in larger system, as an alternative one can adopt **partial ordering** in which of locks need to be always acquired before of locks. Another way to prevent deadlock to occur is to have a **numbering convention** to lock and unlock.
- **Hold-and Wait**: This can be prevented holding all the locks at once atomically. To allow for this to happen, we can put one more lock in front of all the locks that need to be acquired by a thread. However, this leads to problems in case with encapsulation which would need us to know *exactly* which locks need to be acquired beforehand. Also, this would decrease concurrency as all locks must be acquired early on instead of when required.
- **No preemption**: In this, the solution could be to add a which would simply check if the lock needed by a thread is currently held or not, if not, it would return a success code which would then make the lock being acquired by the calling thread, if not it would simply try later to acquire the lock. However, this might lead to another condition called the in which a thread repeatedly fails while trying to acquire a lock, however, this can be resolved by using a timer, that would limit the number of attempts by a thread, thereby decreasing the odds of repeated interference among competing threads.
- **Mutual Exclusion**: One way to avoid mutual exclusion is to make use of lock-free approaches as much as possible. For example, we can use the compare and swap instruction for incrementing counters or for list insertion. These powerful **hardware methods**, can *help avoid the need for explicit locking*.
- Deadlock avoidance is also a solution that we can look at. One way to avoid a deadlock is through scheduling. If we know that a process would need to acquire a certain number of locks, we can schedule in such a way that a deadlock never happens. But such a solution has very limited use-cases.
- Finally, another way is to allow the deadlock to happen. Have deadlock detection in-place, if a deadlock is detected, then the system can be restarted.

CH 36 — I/O devices

- The main takeaways are that, faster the connection needs to be, shorter connection is required.
- A device in general usually has two parts to it, the **interface** and the **internal structure**. The interface usually consists of some kind of mechanism which allows for interaction with the hardware device. The internal structure are responsible for the lower level mechanism that the particular hardware is designed to perform. In such a device, there is usually a protocol that is defined, in which both the interface and the internal consists of some form of hardware for enabling the tasks that they are supposed to perform. For example, the interface might consist of registers such as **Status**, **Command**, and **Data**. In which the status register can be read to see the current status of the device the command register can tell the device to perform a certain task, and the data register can be used to pass/get data to/from the device.
- The protocol has ***four*** steps. In the **first**, the OS waits until the device is ready to receive a command by repeatedly reading the status register, that is, **polling** . **Second**, the OS sends some data down to the data register; When the main CPU is involved with the data movement. **Third**, the OS writes a command to the command register; doing so implicitly lets the device know that both the data is present and that it should begin working on the command. **Finally**, the OS waits for the device to finish by again polling it in a loop, waiting to see if it is finished. It can be noticed that polling might not be a great design decision, because of the overheads associated.
- A better way to go about this is to make use of **hardware interrupts**, that is, when the device completes the execution of the task it was given, it can simply raise an interrupt which would control the flow back to the OS, making it handle the interrupt event. This allow for overlap of computation, meaning that when one particular process is executing a I/O request, the OS can simply switch to another process, thereby improving CPU utilization. **Note that interrupts are not always good**. Because, if the task being performed by the first program is quick, interrupt would add additional overhead as compared polling. Hence**,** if the device is fast, it may be best to poll; whereas if the device is slow, interrupts are better.
- Another technique is **coalescing** in which, the device needing to raise an interrupt can wait a bit before delivering it thereby accumulating other interrupts which could then be processed together by the OS.
- **Explicit I/O** and **Memory Mapped I/O**
- Finally, another important point that needs to be considered is, involving OS with task such as reading and writing to and from a hardware device puts unnecessary overheads, onto OS, which otherwise can be used to run other processes. To solve this, we can have another hardware system in place that would perform this task in place of the OS called the **Direct Memory Access** (DMA). The OS can simply pass the location of the memory to be accessed with the amount/chunk of data that needs to be copied/written to the DMA and then use context switch to switch to another process while the DMA takes over in the background to handle the I/O task. This improves the CPU utilization, and once the DMA is done with the task, it can simply raise an interrupt to inform the OS about it, and then leave it to the OS to either schedule the process that initiated the I/O request or continue with the process that it is currently executing before scheduling the I/O initiating process.

CH 38 — RAID

- **Redundant Array of Inexpensive disk** provide several advantages over a normal disk, them being – **performance**, **reliability**, and **capacity**. It provides all these benefits *transparently*, meaning that to the OS it would still act as any other disk which provides the added benefit of *deployability* which helps to easily replace existing disks with RAID.
- With RAID, the internals are quite complex, it has a **micro controller** that runs firmware to direct the operations of RAID, **volatile memory** (DRAM) to buffer data blocks to be read/written. Alternatively, **non-volatile memory** to buffer writes, an even specialized parity calculation. Finally, **disks** for storage.
- **Fault model** being considered in this chapter is **fail-stop**. Meaning, that a disk can be either working or failed. No intermediate states are possible. Hence when we evaluate the different approaches of building a RAID we will look into three aspects – **Capacity**, **Reliability** and **Performance**.
- **RAID 0 (Striping)**: In this form of RAID, the data is striped across different disks. This striping is based on how much chunk is to be placed on a single disk. RAID 0 provides an upper bound in **performance** and **capacity**. To get the location of the Disk and Offset for the blocks and getting to know where they are stored in the RAID we would use the following equation to get the disk number – (1) and to calculate the offset we use the following equation (2). where A is the logical block address. **Smaller chunk** size would improve parallelism of reads and writes to a single file. While, **larger chunks** would achieve a higher throughput. **RAID 0 Analysis** – Capacity provided is excellent, as it sums to N disks each of size B blocks, which delvers N.B blocks of useful capacity. Reliability is *worst*, it should also be noted that any disk failure would lead to complete data loss. Finally performance is excellent both in terms of Random (**N.R**) and Sequential (**N.S**) workloads. Not a RAID Level at all, since **no Redundancy**.
- **RAID 1 (Mirroring)**: In this form of RAID, there is a mirroring that happens across the disks. This results in a lower **capacity** available for use but higher **reliability** as compared to RAID 0. Also, when considering the performance, writes are slower – (3) where S and R are sequential and random workloads, because the written need to be replicated to another disk, to which the current disk copies. Also, RAID 1, can handle 1 disk failure to up to **N/2 disk failures**, depending on which disk it is. Finally, the read performance depends on what type of read operation is being carried out. If it sequential reads (N/2.S) MB/s, it would not perform any better than other disks, but in case of random reads (better) (N.R) MB/s, the operation can split among the N disks, and hence achieve a higher reading speed. Also, since the writes here are being made to two disk, there is a possibility of something bad happening before both the writes are completed, one way to solve this problem is to make use of a **write-ahead log**, which would keep a log of what the RAID is about to do.
- **RAID 4 (Saving Space with Parity):** In this form of RAID we have an extra disk which would save the parity information pertaining to each of the corresponding blocks. The parity would be calculated by using a simple XOR function which would return 1, if there are even number of 1's and 0 otherwise. Though this method helps in saving space, the tradeoff here is the additional performance that would be incurred because of parity calculation. Also, this parity information can be used to reconstruct data if *at most* one disk is damaged (**Reliability**). The **capacity** associated with RAID 4 is (N-1).B Finally, when considering **performance**, the sequential read performance would be (N-1).S MB/s. In case of sequential writes, RAID 4 can perform full-stripe write which would mean that it would calculate the Parity as soon as it receives the write information, and then parallely write all the information to the disk **(N-1).B** MB/s. Random Reads, would be similar **(N-1).R** to sequential reads, as in it wouldn't consider the parity disk for read performance calculation. Random Writes **(1/2).R** can be performed using two methods, **Additive parity or subtractive parity method**. In the **additive method**, you would read in all the value from the blocks under consideration and then use them to calculate the parity information. This is compute heavy. **Subtractive method** makes use of the block that needs to be overwritten, and compare it with the current information at hand, if they're same the parity block would remain the same, if not the parity bit would have to be flipped, that is, (4) . The problem with RAID 4 is that, even if accessing the disks in parallel is possible, update to the Parity Disk cannot be made parallel, in cases, where small writes are made to different disks. This is known as the **small-write problem.**
- **RAID 5 (Rotating Parity)**: In this form of RAID, it is identical to RAID 4, except for the fact, that the parity information of each block is rotated amongst the different disks. This results in better random read **performance**, as we can now utilize all the disks. Finally, there is a noticeable improvement in random writes as it now allows for parallelism. Random writes happen at **(1/4).R**

(1) $Disk = A \% number\_of\_disks$ (2) $Offset = \frac{A}{number\_of\_disks}$ (3) $(\frac{N}{2} . S)/(\frac{N}{2} . R)$

(4) $P_{new} = (C_{old} \oplus C_{new}) \oplus P_{old}$

CH 39 — Interlude: Files and Directories

- The files in the system have an associated low-level name called the inode number. Just like Files, the Directories also have a low-level name, but its content are quite specific – user-readable name, low-level name. Note that, each entry in a directory either refers to another file or some other directory. One can place directories within directories to form trees, just note that a directory of name **foo** cannot have a file with the same name.
- **Creating Files**: This is done by using the *open()* call, which takes in several different parameters. *O_CREAT* to create file if it does not exist. *O_WRONLY* would be to ensure the file is Write only. *O_TRUNC* would be to truncate the size of the file to zero if it already exists, thus removing any existing content. *S_IRUSR*, *S_IWUSR* would be to associate permissions. One important aspect of *open()* is what it returns – **File Descriptor** which would be integer, private per process which would be used to access the files.
- **Reading and Writing Files**: Using something as simple as the *cat* command to print the content of the files on the terminal requires lots of processing at the backend. **First**, the file would be opened, this would return the file descriptor integer. **Second**, the file descriptor would then be used to read the file, the first parameter would be the FD. Please note that, every file has three open files – **Standard input**, **Standard Output**, and **Standard Error**, hence the FD for a file cannot be 0, 1, or 2. The next parameter would be the buffer that would be used to store the content that would be printed on the screen, and the their parameter would be the buffer size. **Third**, the content would be written on the terminal using the *write* command, which would use the FD as 1, as described above. To which it would write out the content (2nd parameter), the third param would be the size of the content being written. Once complete, the *read* function would be run again, for more content, if not, it would simply return *0* thereby calling the *close* function.
- While reading a file if you need to jump in the buffer to read specific parts or to find something in a file, the *lseek(FD, offset, whence)* system call can be used in which whence is used to describe how the seek [**Not DISK SEEK**] would take place. Hence, once *lseek* is used (to jump to offset 200), and then we use the *read* with the buffer size as set to *50*, we would jump to 250 position in the buffer.
- When writing data to a file, the data is first written out to a buffer, which due to performance reasons would then write the data out to the actual file later. In cases, where this eventual guarantee is not enough **Fsync()** can be used to force the writing immediately..
- Cannot edit the low-level info of directory, it would be done when some file inside the directory is edited. When we **create** a new file, what we are doing is **linking** a file to the directory, and hence to delete the file from the directory we need to call the ***unlink*** function to delete the connection between the file and the directory. When we use the *ln* (**link**) command to link two files, if you edit one of them, the other would have the same content. If you remove one of the files, the other file would still exist. However, if you use **symbolic links**, it would have different inode number, plus if you remove the file that is being linked, the linked file would cease to exist but still appear in the console. Spooky! Size for files linked by symbolic links is **dependent** on the pathname. ∝ pathname
- **Permission bits** – It is represented by 10 bits, the first bit is to show the type of the file *f, d, L* for file, directory and symbolic link respectively. The next 9 bits are for permissions, 3 bits each for the Owner, Group and Others. **Read is specified by 4, Write by 2, and Executable by 1**.

CH 40 — File System (FS) Implementation

- Two basic components of a FS are data structures, and access methods. The on-disk organization of data structure of the FS is as follows. First the disk is divided into **blocks**. Each of these blocks would be of fixed sized. The next thing on a disk is the **user data**, for which space would be reserved. For all the files and directories in the disk, we need a data structure to store information about it so that they can be easily accessed. This information is stored in the **inode** or **inode table**. Next we need information about whether each *inode* or *data region* are free (0) or allocated (1), for this we would use **inode bitmap** and **data bitmap**. Finally, there would be another block that would be allocated to a **Superblock** which would give the information about the number of *inodes* and *data blocks* in the FS. If **superblock corrupt** FS unmountable.
- Inode mainly consists of all the associate metadata that would help describe a file – including but not limited to time of modification, type, size, number of blocks allocated, protection etc.
- Blk = (inumber * sizeOF(inode_t))/ blocksize
- sector =((blk * blockSize)+ inodeStartAddr) / sectorSize. Indirect pointers can be used to point towards aa block with additional pointers. To support even larger files we can have **double-indirect** pointers, triple-indirect pointers etc.
- **Reading a file from the disk** – For this one needs to start at the very start, that is, the root directory, but even for that inode number would be needed, but is already defined and known by the OS, once you've read the inode for the Root, the OS then reads the data content of the root inode, and then it would get the information of the foo directory. Then, the OS would read the content of the Foo inode, to get information about the bar file in the foo directory, for which would have to read the data content of the foo directory, once it has read the bar inode, it would then start reading the data at the different blocks, and then concurrently update the last-accessed value in the bar inode with a write call, this would be repeated until all the data in the bar file has been read.
- **Writing To disk**: Writing to a disk is a much more complex process, which would involve at least 5 I/O operations. First would be read the data bitmap, the second would be to update the data bitmap, the third and fourth world be to read and then write the inode, and finally one would be to write the data content itself. The traversal of the path to the foo directory would be same as above, once there the OS would have to update the inode bitmap with a read and a write call to update the task of creating a file, and then concurrently writing in the foo's data to reflect the file. Then OS would read and write the bar's inode to reflect that the file bar has been created, and reflect his change in the foo inode with the size or any other information. The next steps would be to first read thee bar's inode, and then read and update the data bitmap followed by writing the content to the bar's data blocks, followed by an update to the bar's inode to update the associated metadata. **Caching** and **Buffering, Write Buffering**

CH 41 — Locality and the Fast File System

- The concept of FFS is similar to that described in the previous chapter, the only difference is that the tracks that equidistant from the center across multiple disk would belong to a single Cylinder Group, and each of these groups would have the information about the **inode bitmap**, **data bitmap**, **data**, and the **superblock**.
- FFS allocates directories in the following way. It would store a directory in a cylinder with the least number of directories and a high number of free inodes, and put the directory data and inode in that group. This would help balance directories across groups and also be able to allocate a bunch of files.
- With files, FFS tries to do two things, it would make sure to allocate the data blocks of a file in the same group to avoid longer seek times. Next it places all the files that are in the same directory in the cylinder group of the directory they are in
- One thing to keep in mind is about the large files. In the current aspect of things if we encounter a very large file it would occupy almost the entire group, thus leaving no space for other related files. Hence the solution to this problem would be to divide this data amongst different groups in a way that seek time does not affect the sequential read time. Hence, the way to do this would be select a chunk size that is large enough to amortize the costs associated with the seek of data.
- Finally, FFS introduced a new disk layout, in which sectors were alternatively placed, to allow for more time for the disk to respond to a read request which previously could've been missed if the files were placed in consecutive sectors. Doing this allowed FFS to avoid the additional overhead that would be associated with the rotation of the disk if it had missed the first read. There were other usability improvements as well such as long names (traditional 8 characters). Finally, a new concept was introduced – symbolic links; which allowed the user to create an "alias" to any other file or directory on the system and are much more flexible than the hard links. Also, FFS introduced an atomic *rename()* operation for renaming files.

CH 42 — Crash Consistency: FSCK and Journaling

- The main issue we are focussing over here is how to make the FS consistent with all the crashes happening. The main components that need to be updated when a write happens to the system are the inode bitmap, the data bitmap, inode and the data itself. These are three separate writes, and the system can perform only one write at a time, hence the crash could happen anytime causing the FS to be inconsistent in state.
- Few scenarios are as follows. *Only **one write** goes through*: **Data is written**. In this case, the FS is not in an inconsistent state, because a write did not happen, but the user has lost his data. **Inode is Written**. In this case the inode would point to the disk space, but it would only read garbage value. Further, the data bitmap says that data block is not allocated, but the inode says otherwise. This results in the FS being inconsistent. **Bitmap is Written**. In this case, the FS is inconsistent again, since bitmap says, some data block is allocated, but no inode is pointing to it – could lead to **space leak**. *Only **Two Writes** go through*. **Inode + Bitmap**: In this case, the FS won't be inconsistent, but the data would be garbage. **Inode + Data**: In this the FS would be inconsistent, as the data bitmap, does not recognize that data block is allocated. **Bitmap + Data**: In this case FS would be inconsistent since, data is written and is recognized as being allocated, but no inode is pointed to it, resulting in the FS not knowing which file the data belongs to.
- The first solution proposed is to use a file checker that would check the consistency of the FS *before* the FS is mounted and made available. It would check for several things including: **Superblock sanity checks**: which involved confirming the mapping described in the SB and the actual mappings that FSCK calculated. **Free Blocks**: fsck checks the inode, (double) indirect blocks, to get an overview of block allocation, and build a bitmap and confirms the same with the data and inode bitmaps. If inconsistent, it is resolved by using the info from inodes. **Inode State**: fsck checks for corrupted inodes, and clears them if found and updates the bitmap. **Inode Links**: The link count for each inode is calculated by fsck by building a directory tree, and verified. If a mismatch is found, corrective actions are taken, if allocated node is discovered, but no directory points to it –> *lost and found*. **Duplicates**: if duplicate inodes referring to same block are found, one could be cleared or could be given its own inode. **Bad Blocks**: Pointing outside of the valid range, can be cleared.
    - As can be seen this solution can be very slow, and the performance would be prohibitive when the disk grows in capacity.
- The next solution that we talk about is the Write-ahead logging/**Journaling**. In this all we do is create a journal buffer to hold all the information of the operation that the disk is about to perform, before actually performing. It would consist of a TxB block with a **TransactionID**, to indicate the beginning of the transaction, followed by all the operations, and finally ending with a TxE block, to indicate the end of the transaction, with a **TransactionID**. The operations that are written to this journal block, are then written to the disk one by one, and not together as that could lead to problems, as it would be susceptible to disk scheduling, and a crash could lead to missing out on some transaction. To avoid this problem, the disk would first write all the blocks except the TxE block using what is called **checkpointing**, which would be to write out he contents of the update to the final disk location. Hence the steps that would be involved are – **Journal write**, **Journal commit**, and **Checkpointing**. If the file system crashes at any point, the steps would be re-done (**redo logging**).
- **Limiting the journal**: Journal size should be limited because, if not then the amount of tasks that the journal would hold, that would need checkpointing would grow to a point that it would no longer be able to hold transactions, and even in case of a crash, the recovery would take a lot of time. To address this, we would make the journal circular and adding a **Journal Superblock** which would keep information about the tasks that haven't been check-pointed yet, this in turn would help clear out the tasks that have been checkpoints so that the journal could be cleared for newer tasks to be added.
- **Metadata Journaling**: The approach here (**data/ordered journaling**) is a bit different as what is followed above, in way that the data being written to the disk is no longer a part of the journaling process, and is written before the journaling process even begins. This reduces the cost of the double writing (first to the journal and then again to the disk). The data is written **before** the journalling, process as otherwise the inode could be pointing to garbage values. (**Data Write**, **Journal Metadata Write**, **Journal Commit**, **Checkpoint Metadata**, **Free**)
- **Backpointer-based consistency** makes sure that the data block being pointed by the inode, has a back pointer to the inode. The FS can be sure of consistency if both the forward-pointer and the back pointer are in tandem.

CH 43 — Log-Structured File Systems

- Ideal Filesystem would hence use sequential disk **+** focus on increasing the write performance **+** it would work on performing well on common workloads **+** work well with raids
- Log-structured File System
- It would first buffer all the updates + metadata in an in-memory segment. Once the memory segment is full, the content is written to disk in one long sequential part of the disk. **NOTE** : LFS never over-writes existing data, but rather always writes segments to free locations.
- Writing to disk Sequentially
- The basic idea is to load all the data blocks and the corresponding meta-data blocks including the inode blocks in the in-memory segment before writing the data down in a sequential manner.
- Assume a single data block to be written, once it is written in the in-memory block, the corresponding inode block would be written right next to it, while it would have a pointer which would point back to the data block. The data block is usually 4KB in size, whereas inode is around 128 bytes in size.
- Writing to disk sequentially and efficiently
- The basic idea is to use the concept of **write buffering** which makes use of an in-memory buffer for buffering the writes first, and once the buffer is full, then sequentially write out the content of the in-memory segment to the disk.
- For example if we have data blocks 0,1,2,3,4 and the corresponding inode blocks. Then we would first write down all the data blocks, then all the inode blocks, and then create a link back from the inode blocks to the corresponding data blocks.
- How much to buffer?
- The data that would be needed to be buffered for optimal performance would depend on the disk
- The factors it would depend on are: Peak transfer rate (P), Time it takes to position the head (T), The effective bandwidth of peak (F),that is Fraction(F) of Peak Rate Where D is the data that can be buffered for optimal performance
- **Inode map** – Since, inode in LFS is scattered all throughout the disk, we need an inode map takes the inode number as input and produces the disk address of the most recent version of the inode. Any time an inode is written the imap is updated. And the imap is written write next to the newly edited/updated data
- **Checkpoint region** –The checkpoint region contains the pointers to the latest pieces of the inode map and thus the inode map pieces can be found by reading the CR first. The CR is updated every 30 seconds or so to avoid it affecting the performance
- **Reading a file** – So what happens is first the OS would load up the CR and search for inode map. Once the imap is loaded, it would locate the latest version of the inode using the inode number that it received for the file that it wants to read. Once it has the inode number and the has loaded the inode it would directly load up to that position in the disk because the inode would be pointing back towards the data block to be read. Writing to a file is similar, also with inode map, LFS avoids the **Recursive Update Problem**
- **Garbage collection** – A noticeable problem with LFS is that though it works on improving the write performance by continuously updating the files in the system, the older versions of the files are still in the filesystem which are scattered. These are the garbage values. One could argue that since, these files are nothing be older versions of the same file, a versioning system could be created to allow the user to restore the older version of the files. However, the problem is that LFS only keeps one live version of the file, and the older versions are truly garbage now, since they cannot be recovered. Hence, something called garbage collection should be performed to collect all the older version of all the files in the system and cleared out, so that the file system can be made free for other purposes.
- However, the garbage cleaner cannot just move out on a cleaning spree, as that could lead to creation of free holes in the system. Hence, the LFS aims to go segment by segment thus clearing up large chunks of space for subsequent writing
- There would be a **segment summary block** which would be used to determine the liveness of a block, if the map in the SS points exactly to the exact address on the disk, it would be a live block, else it would be a dead block.
- It is better to clean out cold segment firsts, than to clean out hot segments, which would be used more frequently. Also to enable recovery during crashes, LFS keeps two **Checkpoint Regions** and updates them alternatively, if LFS sees an inconsistent paid of timestamps between the two CR

$$T_{write} = T_{position} + \frac{D}{R_{peak}} \qquad R_{Effective} = \frac{D}{T_{position} + \frac{D}{R_{peak}}}$$

$$D = \frac{F}{1-F} * R_{Peak} * T_{Position}$$

CH — 44 Flash Based SSD

- Flash chips are organized into banks or planes, accessed in two different sized units – blocks and pages. Within each bank there are large number of blocks, which in turn have a large number of pages.
- Flash operations – **Read**: Can access any location uniformly. **Erase**: Before writing to a page, you need ERASE the entire block (settles to 1). Remember to copy important content, before erasing. **Program**: Change bits from 1s to 0s. Can also cause some neighboring bits to flip – **read disturbs / program disturbs**.
- Flash Translation Layer provides the control logic need for orchestrating device operations. It would take the read and write requests on logical blocks, and translate them into low-level read, erase, and program commands. Another goal is to maintain high performance and reliability, and it can be done by reducing **write amplification**, that is, write issued by FTL / write issued by client.
- **Log-Structured FTL**– In this a write to a block would be appended to the next free spot available to the currently-being written block. To allow for reads, a mapping table is made. The drawback of this form of FTL is that it leads to Garbage, and periodically, garbage collection needs to be performed. **Garbage Collection**: Read in live pages from the block, write those live pages the log, & then erase the block. To determine the block liveness, one can refer the table mapping.
- Page Level mapping can be very space consuming, alternatively we can use Block-Level Mapping(BLM), but there are some performance tradeoffs. In BLM we would need a **chunk number** and an **offset**. Chunk Number would point at the block, and then offset can be used to map into the page we're looking for.
- **Hybrid Mapping**: It has a page level mapping called **log table** and a per-block mapping called the **data table**. **Switch merge** –> convert log table and data table mappings into one data table reading. **Partial merge** –> Have both log table and data table values.
- **Wear Leveling** – Spread the writes throughout the SSD. Log-Structured FTL does a good job, but for long-lived data FTL must periodically re-write the content elsewhere, making a block available again.

CH 45 Data integrity and protection

- Disk failure – Latent Sector errors (LSEs) and block corruption. **LSE**: In this the sectors of the disk would be damaged in some way. The disk would deploy some error correcting code to correct the error. Similarly a block might get **corrupted**, and this would be considered as silent faults since they would be hard to detect, as the disk would not indicate any problem.
- To solve LSE, some redundancy mechanism can be used to reconstruct the lost data with RAID where in parity information can be used or the mirrored information can be used to reconstruct the data lost.
- To detect corruption, checksum for the data can be calculated to detect the corrupt data, and then recovery as before can be used to reconstruct the lost data. The input would be the data, and the output would be the summary of data, for example, when XOR is calculated over the data, if on later access, the checksum calculated before the corruption and after the corruption would be different, this would help detect the corruption. Ways of calculating checksum, XOR, Addition, Fletcher's algorithm, Cyclic Redundancy Check. Checksum can either be stored along with the data, (Update needs 1 R and 1 W), whereas it can also be allocated a new block for N data blocks – (Update needs 1 R, 2 W)
- The detection of corruption using checksum usually happens during the read, when the client reads the checksum stored, and compares it with the checksum that it would calculate. If they're different that's when the client knows that the data is corrupted. To identify **Misdirected Writes** we can add a physical identifier to the checksum. Checksum however, cannot help with **lost writes**, other mechanism such as **write verify** or **read-after-write** can ensure that no writes are lost.
- Disk also deploys **disk scrubbing** a mechanism that would periodically read through every block of the system and verify the checksum.
- Checksum also induces **overheads** specifically with **space** and **time**. Time: overhead can be reduced by combining the aspect of copying the data (kernel cache to user buffer) and calculating the checksum.

CH – 49 NFS

- Focus – Transparency, Performance, and simple + fast crash recovery.
- Solution is to use steels protocol for transfer of information. In a stateless protocol, the client would send all the information that would be necessary to carry out an operation.
- In case of a stateful, protocol, if the server crashes, the server will have to run some kind of recovery to be able to understand subsequent client requests. Which adds to the overhead.
- **NFSv2**: File handle is how the server sends the client the information needed for access to a particular file. It consists of three identifiers. **Volume ID** – It helps to identify the file system/partition. The **inode no**. – helps identify the particular file. And **Generation number** is needed when reusing an inode. The client needs to obtain the File Handle from the server using the LOOKUP (one request per level in the DIR a/b/c.txt will need 3 requests.) protocol, and the connection is established with the MOUNT protocol. Once the client has the File handle, it can issue the READ and WRITE commands to the server.
- In case of a server crash, the NFSv2 protocol, just **retries** the sent requests, if it receives no response from the server, this possible because the requests are **idempotent**, meaning that the effect of performing the same operation multiple times is equivalent to the effect produced when the request is processed once. READ, WRITE, GETTATR, etc are idempotent, but CREATE or MKDIR is not.
- The client uses **caches** for for caching the file data and metadata. Thus, while the first access is expensive, subsequent accesses are serviced quickly out of memory. The client also uses the cache for buffering writes before actually sending the **WRITE** request to the server, that is, it would decouple the writes to the server allowing for an **immediate success** being returned to the application. This could lead to **cache inconsistencies** when multiple clients are involved – **update visibility** when the client gets old copy and is solved by **flush on close**/**close to open**, and **stale cache**, when the client is working on a version of a file that is old, to solve this, the client would first send a GETATTR request to confirm whether it has the latest copy before accessing the file. But this could lead the server to be overwhelmed with GETATTR requests, hence an **attribute cache** was used with a timeout value, to update the attribute periodically.
- Also to avoid inconsistencies, the server must commit each write to a stable storage before informing the client of success.

Virtual Machine Monitor

- **Virtualizing CPU**: In this, the basic technique that is used is limited direct execution. A virtual machine monitor must perform a **machine switch** between running virtual machines. Thus, when performing such a switch, the VMM must save the entire machine state of one OS (including registers, PC, and unlike in a context switch, any privileged hardware state), restore the machine state of the to-be-run VM, and then jump to the PC of the to-be-run VM and thus complete the switch.
- The VMM cannot execute in kernel mode, but when the OS it is hosting first boots up, it would record the location of the trap handler of the hosted OS. Now, when the VMM receives a trap from a user process running on the given OS, it knows exactly what to do: it jumps to the OS's trap handler and lets the OS handle the system call as it should.
- The OS on the VMM, cannot be running in kernel mode either, it runs on a special mode called **supervisor mode**, which allots it extra memory than usual, which is used by the OS for its data structures. Hence when a process on the hosted OS makes a **(1)** System Call. **(2)** The the process is Trapped, and the VMM calls the OS trap handler. **(3)** The actual OS' Trap Handler would respond to the trap, by decoding the trap and executing the system call & when it is done, it would issue a return from Trap to go back to the VMM. **(4)** The VMM would then do the real return from trap to the process on the hosted OS. **(5)** Which would then finally, resume execution.
- To allow for virtualizing of memory, another layer of virtualization needs to be added so that multiple OSes can share the actual physical memory. This extra layer of virtualization makes "physical" memory a virtualization on top of what the VMM refers to as **machine memory**. Each OS maps virtual-to-physical addresses via its per-process page tables; the VMM maps the resulting physical mappings to underlying machine addresses via its per-OS page tables.
- The steps are as follows. **(1)** When a process on the hosted OS Loads something from memory, it would be a TLB miss, which would result in a trap. **(2)** This trap is handled by the VMM which would call into the actual OS' TLB handler. **(3)** The OS TLB miss handler would extract the VPN from the VA, do a page lookup and get the PFN, if valid and update the TLB. **(4)** The VMM would now update the TLB because the OS, during "boot", tried to install its own trap handlers. The OS TLB miss handler then runs, does a page table lookup for the VPN in question, and tries to install the VPN-to-PFN mapping in the TLB. However, doing so is a privileged operation, and thus causes another trap into the VMM. At this point, the VMM plays its trick: instead of installing the OS's VPN- to-PFN mapping, the VMM installs its desired VPN-to-MFN mapping. After doing so, the system eventually gets back to the user-level code, which retries the instruction, and results in a TLB hit, fetching the data from the machine frame where the data resides. **(5)** These per-machine page tables need to be consulted in the VMM TLB miss handler in order to determine which machine page a particular "physical" page maps to, and even, for example, if it is present in machine memory at the current time, to reduce the cost associated with TLB misses, a software TLB was introduced. **(6)** The return from trap in the previous results in the VMM returning a return from trap to the hosted OS, which then **(7)** resumes execution.
- The VMM first consults its software TLB to see if it has seen this virtual-to-physical mapping be- fore, and what the VMM's desired virtual-to-machine mapping should be. If the VMM finds the translation in its software TLB, it simply installs the virtual-to-machine mapping directly into the hardware TLB, and thus skips all the back and forth in the control flow above
- When a VMM is running underneath two different OSes, one in the idle loop and one usefully running user processes, it would be useful for the VMM to know that one OS is idle so it can give more CPU time to the OS doing useful work.
- **Demand Zeroing**: The reason for doing so is simple: security. If the OS gave one process a page that another had been using *without* zeroing it, an information leak across processes could occur, thus potentially leak- ing sensitive information. Unfortunately, the VMM must zero pages that it gives to each OS, for the same reason, and thus many times a page will be zeroed twice, once by the VMM when assigning it to an OS, and once by the OS when assigning it to a process.

CH 50 – AFS

- The main focus of AFS was to introduce **scale** into a distributed file system such that a server can support as many clients as possible. Also, AFS is very different from NFS, such that in NFS cache consistency is hard to describe since it depends directly on low level implementation details including client side cache timeout intervals. In AFS cache consistency is simple and readily understood: when a file is opened a client will *generally* get the latest consistent copy from the server.
- **AFS Version 1** : In this version, when an *Open()* call is made to the server, with the entire filename being sent to the server, the server sends the entire file to the client, which is then ***written*** to the local disk of the client. Any subsequent *Read()* or *Write()* functions are redirected to the **local disks** thereby increasing the efficiency and avoiding the network transfer of files and the content. Once the operation is completed *AFS Client* would check if the file was modified, and if so it flushes the new version back to the server with the path and a ***Store*** *protocol message*. Next time the file is accessed by this client, it would use the ***TestAuth*** *protocol message* in order to determine whether the file has been updated, since the last time the client accessed it. If so it would fetch and store the file in the client local disk, if not the client would directly access the cached version of the file, thereby avoiding the network transfer. Only Files **not** directories.
- **Problems with version 1**: **(1)** – **Path traversals were costly**: When performing the Fetch and Store protocol request, the entire path traversal sent by each of the client made the server spend too much of its CPU time simply walking down directory paths. **(2)** – **Too many TestAuth messages**: Like in NFS with GETATTR, AFSv1 generated a large amount of traffic only to check whether a local file was valid or has been modified, Thus, this resulted in there servers spending too much of their time on telling the clients whether it was *OK* to use the local cached copy of the file. **(3) Load was not balanced across Servers**: Solved with Volumes. **(4)** The server used a single distinct process per client thus inducing context switching and other overheads. All these problems resulted in limiting the scalability of the system.
- **AFS Version 2**: A notion of **callback** is introduced. In this the client assumes that a file is valid until the server tells it otherwise. This is to reduce the TestAuth message protocol crowding on the server side. Also, AFSv2 introduced the notion of file identifier which includes – **volume identifie**r, **file identifier** and a ***uniquifier*** which enabled the reuse of the volume and file IDs when a file is deleted. Thus instead of sending the entire path for a file the client would walk the pathname one piece at a time caching the results and reducing the load on the server. The key difference from NFS is that with each fetch of a path the AFS client would establish a callback with the server
- **Cache Consistency**: Because of callbacks discussed in the previous point, the notion of cache consistency is simplified. There are two important cases to consider – Consistency between processes on different machines, and consistency between processes on the same machine. **On Different Machines** –When a client gets a copy of the file, it would update the visibility and invalidate any cached copies, once the update (if any) is completed to the file, then the connection with server is broken for callbacks. When the client wants to restart working on the file it would initiate a new callback connection. This step ensures that that client will no longer read stale copies of the file and subsequent opens on those clients will require a re-fetch of the file. Thereby making a consistent copy of the latest version. **On Same Machine** – in this case writes to a file are immediately made visible to other local processes. **Also there is a notion of last writer wins**. Meaning that if there are multiple clients editing the same file, the client which closes the file last would be the one whose changes would be saved to the AFS permanent storage.
- **Performance of AFSv2**: **First**, in many cases, the performance of each system is roughly equivalent. **Second**, difference arises during a large-file sequential re-read (Workload 6). Because AFS has a large local disk cache, it will access the file from there when the file is accessed again. NFS, in contrast, only can cache blocks in client memory; as a result, if a large file (i.e., a file bigger than local memory) is re-read, the NFS client will have to re-fetch the entire file from the remote server. **Third**, we note that sequential writes (of new files) should perform similarly on both systems (Workloads 8, 9). AFS, in this case, will write the file to the local cached copy; when the file is closed, the AFS client will force the writes to the server, as per the protocol. NFS will buffer writes in client memory, perhaps forcing some blocks to the server due to client-side memory pressure, but definitely writing them to the server when the file is closed, to preserve NFS flush-on-close consistency. However, realize that it is writing to a local file system; those writes are first committed to the page cache, and only later (in the background) to disk, and thus AFS reaps the benefits of the client-side OS memory caching infrastructure to improve performance. **Fourth**, we note that AFS performs worse on a sequential file over- write. Overwrite can be a particularly bad case for AFS, because the client first fetches the old file in its entirety, only to subsequently over- write it. NFS, in contrast, will simply overwrite blocks and thus avoid the initial (useless) read. **Finally**, workloads that access a small subset of data within large files perform much better on NFS than AFS. In these cases, the AFS protocol fetches the entire file when the file is opened; unfortunately, only a small read or write is performed. Even worse, if the file is modified, the entire file is written back to the server, doubling the performance impact. NFS, as a block-based protocol, performs I/O that is proportional to the size of the read or write.
- **Other Improvements**: **(1)** AFS provides a true global namespace to clients, thus ensuring that all files were named the same way on all client machines. NFS, in contrast, allows each client to mount NFS servers in any way that they please, and thus only by convention would files be named similarly across clients. **(2)** AFS also takes security seriously, and incorporates mechanisms to authenticate users and ensure that a set of files could be kept private if a user so desired. NFS, in contrast, had quite primitive support for security for many years. **(3)** AFS also includes facilities for flexible user-managed access control. Thus, when using AFS, a user has a great deal of control over who exactly can access which files. NFS, like most UNIX file systems, has much less support for this type of sharing. **Finally**, as mentioned before, AFS adds tools to enable simpler management of servers for the administrators of the system. In thinking about system management, AFS was light years ahead of the field.

Workload	NFS	AFS	AFS/NFS
1. Small file, sequential read	$N_s \cdot L_{net}$	$N_s \cdot L_{net}$	1
2. Small file, sequential re-read	$N_s \cdot L_{mem}$	$N_s \cdot L_{mem}$	1
3. Medium file, sequential read	$N_m \cdot L_{net}$	$N_m \cdot L_{net}$	1
4. Medium file, sequential re-read	$N_m \cdot L_{mem}$	$N_m \cdot L_{mem}$	1
5. Large file, sequential read	$N_L \cdot L_{net}$	$N_L \cdot L_{net}$	1
6. Large file, sequential re-read	$N_L \cdot L_{net}$	$N_L \cdot L_{disk}$	$\frac{L_{disk}}{L_{net}}$
7. Large file, single read	L_{net}	$N_L \cdot L_{net}$	N_L
8. Small file, sequential write	$N_s \cdot L_{net}$	$N_s \cdot L_{net}$	1
9. Large file, sequential write	$N_L \cdot L_{net}$	$N_L \cdot L_{net}$	1
10. Large file, sequential overwrite	$N_L \cdot L_{net}$	$2 \cdot N_L \cdot L_{net}$	2
11. Large file, single write	L_{net}	$2 \cdot N_L \cdot L_{net}$	$2 \cdot N_L$

CH 37 – Hard disk drives

- **Geometry** – Hard-disks have one or more circular hard **platters**, each of which has 2 **surfaces**. All the platters in the hard-disk are bound together around the **spindle**, connected to a motor. Data is encoded in each of the concentric circles of **sectors** called **tracks**. The reading/writing of data on the disk is accomplished by a **disk head**, one for each surface and is attached to a single **disk arm**.
- **Rotational Delay** – the delay which is caused when the disk head (while constant in the sector) waits for the particular track to come under it, while the disk rotates. **Seek Time** – it is the delay when multiple sectors are under consideration, and the head needs to move from one track of one sector, to another track on another sector. This would result in additional delay, called seek time. The seek consists of – *acceleration*, *coasting(full speed)*, and *deceleration;* finally followed by *settling* (carefully positioning the head on the track req.) Once positioned, the data can be transferred. Tp allow for lesser seek time, and fewer rotational delays, the sectors are *skewed*.
- Also, there is an additional component – **Cache/ track buffer**, for holding data. This enables two types of caching – **write back** and **write through**. With write back, the acknowledgement for the write is given immediately after the data is put on the disk, where as with write through, the acknowledgment is given after the write has been actually put on the disk.
- **Disk Scheduling** – **Shortest Seek Time First** (SSTF) – In this disk scheduling form, the shortest time to find and position on the track is taken into consideration for scheduling first. However, the problem is that the **disk geometry** is not available to the OS, hence it can only implement the nearest block first algorithm. The second con is **starvation**, if the requests come in for blocks next to each other, the blocks farther apart wouldn't be serviced.
- **SCAN** – In this a continuous movement from the outer tracks to the inner tracks and in reverse order is made, (**sweep**) and all the requests along the line are processed. **F-SCAN** puts all the new requests at the back of the queue. Circular SCAN (**C-SCAN**) unlike SCAN, resets the head to Outer track after one sweep, thereby not favoring the middle tracks. **Shortest Time to Positioning First**

$$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer} \qquad R_{I/O} = \frac{Size_{transfer}}{T_{I/O}}$$

CH40	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
open(bar)			read			read				
				read			read			
					read					
read()					read					
					write			read		
read()					read					
					write					read
					read					
					write					

Figure 40.3: **File Read Timeline (Time Increasing Downward)**

```
typedef struct __lock_t { int flag; } lock_t;
void init(lock_t *mutex) {
// 0 -> lock is available, 1 -> held
mutex->flag = 0;
}
void lock(lock_t *mutex) {
while (mutex->flag == 1) // TEST the flag
                ; // spin-wait (do nothing)
mutex->flag = 1;  // now SET it!
}
void unlock(lock_t *mutex) {
mutex->flag = 0;
}
```

1) First Attempt: A Simple Flag

```
void lock(lock_t *lock) {
   while (1) {
      while (LoadLinked(&lock->flag) == 1)
         ; // spin until it's zero
   if (StoreConditional(&lock->flag, 1) == 1)
         return; // if set-it-to-1 was a success:
all done
                 // otherwise: try it all over again
                 }
}
void unlock(lock_t *lock) {
   lock->flag = 0;
}
```

5) Using **(4)** to build a lock

```
int TestAndSet(int *old_ptr, int new) {
      int old = *old_ptr; // fetch old value at
old_ptr
      *old_ptr = new;     // store 'new' into
old_ptr
      return old;       // return the old value
}
```

2) Test & Set

```
int CompareAndSwap(int *ptr, int expected,
int new) {
   int actual = *ptr;
   if (actual == expected)
      *ptr = new;
   return actual;
}
```

3) Compare and Swap

```
int FetchAndAdd(int *ptr) {
   int old = *ptr;
   *ptr = old + 1;
   return old;
}
```

6) Fetch and Add

```
int loops;
cond_t cond;
mutex_t mutex;

void *producer(void *arg) {
int i;
for(i=0; i<loops; i++){
 Pthread_mutex_lock(&mutex);
 while (count == 1)
  Pthread_cond_wait(&cond, &mutex);
 put(i);
 Pthread_cond_signal(&cond);
 Pthread_mutex_unlock(&mutex);
 }
}
void *consumer(void *arg) {
int i;
for (i=0;i<loops;i++){
 Pthread_mutex_lock(&mutex);
 while (count == 0)
  Pthread_cond_wait(&cond, &mutex);
 int tmp = get();
 Pthread_cond_signal(&cond);
 Pthread_mutex_unlock(&mutex);
 printf("%d\n", tmp);
 }
}
```

7) Producer Consumer: Single CV & While

```
int LoadLinked(int *ptr) {
   return *ptr;
}
int StoreConditional(int *ptr, int value) {
   if (no one has updated *ptr since the
LoadLinked to this address) {
                *ptr = value;
                return 1; // success!
}
else{
      return 0; // failed to update
   }
}
```

4) Load Linked and Store Conditional

```
typedef struct __lock_t {
   int ticket;
   int turn;
} lock_t;

void lock_init(lock_t *lock) {
   lock->ticket    = 0;
   lock->turn      = 0;
}

void lock(lock_t *lock) {
int m = FetchAndAdd(&lock -> ticket);
while (lock->turn != m)
; //spin
}

void unlock(lock_t *lock) {
lock->turn = lock->turn + 1;
}
```

7) Ticket Locks