Intro, Threads & Concurrency

- What are the three views of an operating system? Describe them.

Application View - provides an execution environment for running programs
System View - manages the hardware resources of a computer system
Implementation View - OS is a concurrent, real-time program

- Define the kernel of the OS. What things are considered part of the OS but not the kernel?

Kernel: The OS kernel is the part of the OS that responds to system calls, interrupts and exceptions.
OS: The OS as a whole includes the kernel, and may include other related programs that provide services for applications. e.g. utility programs, command interpreters, programming libraries.

- What are some advantages of allowing more than one thread to exists simultaneously?

1. programs can run faster
2. if one thread has to block, another may be able to run

- What is contained in a thread context?

Each thread has its own private register contents and stack, and they share access to the program's global variables and heap.

- Describe what happens during a context switch? What is dispatching?

During a context switch:

1. decide which thread will run next(scheduling)
2. save register contents of current thread
3. load register contents of next thread

- What is the difference between thread_yield() and thread_sleep()?

thread_yield: running thread voluntarily allows other thread to run
thread_sleep: the running thread blocks,waits for some particular event to wake it up
thread preempted: running thread involuntarily stops running

- How is an involuntary context switch accomplished? Define quantum.

Preemption means forcing a running thread to stop running so that another thread can have a chance. This is normally accomplished using interrupt.
Quantum: an upper bound on the amount of time that a thread can run.

- Describe what happens when an interrupt occurs. Where do interrupts come from?

Interrupts are caused by system devices(hardware). e.g. a timer, a disk controller

When an interrupt occurs, the hardware automatically transfers control to a fixed location in memory, and call interrupt handler:

1. create trap frame to record thread context at the time of the interrupt
2. determines which device caused the interrupt and performs device-specific processing
3. restores the save thread context from the trap frame and resumes execution of the thread

Key point:
Implementing Concurrent Threads:

1. P processors, C cores per processor, M multhreading degree per core => PCM threads can execute **simultaneously**
2. timesharing - rapidly switch

- The preempted thread changes the state from running to ready, and it is placed on the ready queue.

Thread_switch (high-level context switching code):
1. Finds the next thread to run
2. Calls switchframe_switch to perform the low-level context switch

- Thread_switch(caller) save the t-registers prior to calling switchframe_switch
- switchframe_switch(callee, generate swtichframe, which should be drawn on stack)
1.save s-registers which data is from OLD thread, and
2.load NEW thread data into s-registers
3.pop the switchframe stack from off of the NEW THREAD's stack.
4.jump to NEW thread thread_switch return address

Context Switch can be caused by: Thread_yield, Thread_exit, Wchan_sleep, Preemption.

Synchronization

Keyword *volatile*:
- Value in memory is NOT accurate
- Compiler does NOT optimize value into register
- *volatile* forces the compiler to load and store the value on every use
- Does NOT provide synchronization; ONLY turns off that optimization to prevent these optimizations from causing race conditions

- Race conditions caused by multiple threads reading/writing same variable OR compiler re-arranging/optimizing code OR CPU re-arranging loads/stores

- Multi-threaded code requires **sequential consistency** for shared variables
○ I.e., load/store order is same for optimized and non-optimized code

- To prevent race conditions, we can enforce **mutual exclusion** on critical sections in the code.
- any primitive type value insides mutual exclusion should be volatile to prevent race condition!

- Hardware-Specific Synchronization:
Test and Set Examples:
x86: xchg src, addr
SPARC cas: cas addr, R1,R2
MIPS: load-linked and store-conditional
LL returns the current value of a memory location, where subsequent SC to the same memory location will store a new value only if no updates have occurred to that location since the load-linked

- **Locks**:
Spinlocks **spin**(busy waits), Locks **block**.
When a thread blocks, it stops running:
1. the scheduler chooses a new thread to run
2. a context switch from the blocking thread to the new thread occurs
3. the blocking thread is queued in a wait queue(NOT ready list)
Eventually, a blocked thread is signaled and awakened by another thread

- **Wait Channels**
Wchan_sleep: * blocks calling thread on wait channel
* causes a **context switch**, like Thread_yield
Wchan_wakeall: unblock **all** threads sleeping on wait channel
Wchan_wakeone: unblock **one** thread sleeping on wait channel
Wchan_lock: prevent operations on wait channel - No wakeup on wait channel before Wchan_sleep get called

- **Semaphores**
  - What is a semaphore? Which two operations does it support? Are they atomic?

A semaphore is a synchronization primitive that can be used to enforce mutual exclusion requirement.
A semaphore is an object that has an integer value(let's call it sem), and that supports two operations:

- **P**: if sem > 0, decrement the value. Otherwise, wait until sem > 0
- **V**: increment sem

P and V are *atomic*

- What is the primary difference between a lock and a binary semaphore?

- A **binary** semaphore is just a semaphore with **one** resource, it protects a critical section much like a lock but with NO concept of ownership - which means the resource can be released by anybody. * Inability to solve priority inversion problem
- A normal semaphore lets a number of threads in, it's for resource counting

- **Condition Variables**
  - What is a condition variable? How are they used in conjunction with locks?

CV is a synchronization primitive. Each CV is intended to work together with a lock.

CVs are **only** used from within the critical section that is protected by the lock

- Note: signalling\broadcasting to a CV with no waiters has NO effect

The signal should happen inside of the critical section **protected by the lock**.

Hence, the signalling thread should be the **owner** of the lock passed in.

- Deadlock:

A deadlock is a situation wherein two or more competing actions are each waiting for the other to finish.

- Can deadlock occur with only one thread?

No, def is above.

Two techniques for Deadlock Prevention:

1. No Hold and Wait: A thread must make a single request for all of the resources.

   Advantage:

   ● Threads **never** own locks while they are blocked waiting to acquire another thread

   Disadvantage:

   ● Threads spin while trying to repeatedly acquire a set of locks

2. Resource Ordering: Order the resource types, and require that each thread acquire resources in increasing resource type order.

Processes and the Kernel

- What is a Process?

A process is an environment in which an application program runs.

- What information is contained within a process?

- one or more threads

- virtual memory, used for the program's code and data

- other resources, e.g. file and socket descriptors

* Processes are created and managed by the kernel.

* Each program's process **isolates** it from others programs in other processes

- Can threads from one process access memory from another process?

No, see above.

Fork:

- *fork* creates a new process(the child) that is a **clone** of the original(the parent)
- after *fork*, both parent and child are executing copies of the same program
- virtual memory of parent and child are identical at the time of the fork, but may diverge afterwards
- *fork* is called by parent, but returns in both the parent and the child

  * parent and child see different return values from *fork* (parent: get child pid, child: get 0)

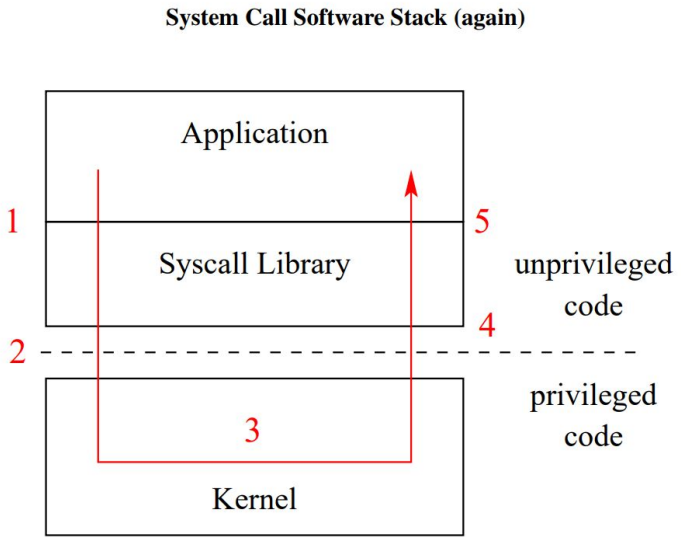
- *_exit* terminates the process that calls it
- process can supply an exit status code when it exits.
- kernel **records** the exit status code in case another process asks for it(via *waitpid*)
- Parent CAN DIE before Child
- *waitpid* let a process **wait** for another to terminate, and **retrieve** its exit status code
- Only PARENT can call *waitpid* on CHILD(NO grandchild)
- *execv* changes the program that a process is running
- the calling process's current virtual memory is destroyed

- The process gets a **new** virtual memory, initialized with the code and the data of the new program to run
- After *execv*, the **new** program **starts** executing

System Calls:

- System calls are the interface between processes and the kernel.

- A process uses system calls to request operating system services.



Kernel code runs at a **higher** level of ***execution privilege*** than application code

- privilege levels are implemented by the CPU

Application programs CANNOT directly call kernel functions or access kernel data structures

- What is an exception? What is the difference between an exception and a system call? And exception and a interrupt?
- How does a program make a system call?
- Only two things that make kernel code run

1. **Interrupts**

   * interrupts are generated by devices(hardware)

2. **Exceptions**

   * exceptions are caused by instruction exception(software)

- What can the kernel do in privileged execution mode that user programs can not directly accomplish?
- Handle interrupts and exceptions

- Interrupt handlers are part of the kernel

- Exception handler is part of the kernel

- Exceptions are detected by the CPU during instruction execution

- When an interrupt/exception occurs, the processor switches to privileged execution mode

E.g.: system call: fork, exception handler in kernel: sys_fork

How System Calls Work?

- Application calls library wrapper function(syscall) for desired syscall

- The kernel's exception handler checks the syscall codes to determine which syscall has been requested.

- Syscall codes must be known by both the kernel(check which syscall should be requested and application(store into specific register).

- System Call Parameters: On the MIPS:

1. parameters go in registers a0-a3
2. result success/fail code is in a3 on return
3. return value or error code is in v0 on return

a3: 0(success) => v0: return value

a3: 1(fail) => v0: error code

User and Kernel Stacks

- Every OS/161 process thread has two stacks, **User Stack and Kernel Stack**, although it only uses one at a time
- **User(Application) Stack:** used why application code is executing

  * the stack is located in the application's virtual memory

  * the **kernel** creates this stack when it sets up the virtual address memory from the process

- **Kernel Stack:** used while the thread is executing kernel code, after an exception or interrupt

  * this stack is kernel structure

  * this stack also holds **trap frames** and **switch frames**(cuz the kernel creates trap frames and switch frames)

Exception Handling in OS/161

* *mips_trap* gets called after *trap frame* was created

- The mips trap function is called to handle system calls, exceptions, and interrupts. How does it differentiate between these three cases?
- *mips_trap* determines what **type** of exception this is by looking at the exception code(interrupt? system call? address translation exception? something else?)

- When returning from a system call, the program counter is not automatically advanced by the hardware. Why is this?
- The program counter for the user process is on the trapframe, and it was saved at the time of an exception/interruption. The kernel that handles the syscall doesn't use the same PC

Virtual Memory: Pre-Midterm:

Physical Addresses:

- A system uses physical address and virtual addresses. How many physical address spaces are there in a computer?

  If physical addresses have P bits, the max amount of addressable physical memory is 2^P bytes.(Virtual address spaces is similar)

- The actual amount of physical memory on a machine may be less than the max amount that can be addressed

Virtual Addressees:

- The kernel provides a separate, private virtual memory for each process

- The VM of a process holds the code, data, and stack for the program that is running in that process

  - How many virtual address spaces?

    If virtual addresses are V bits, the max size of a VM is 2^V bytes

- Running applications see only VM

- Each VM is mapped to a different part of physical memory

Address Translation:

- Address translation is performed in hardware, using information provided by the kernel

Address Translation fro Dynamic Relocation:

- CPU includes a **memory management unit(MMU)**, with a **relocation register** and a **limit register**

  - relocation register holds the physical offset(R) for the running process' virtual memory
  - limit register holds the size L of the running process' virtual memory

- Translation is done in hardware by MMU

  - In dynamic relocation, when does the value of the relocation register in the MMU change?
  - The kernel maintains a separate R and L for each process, and changes the values in the MMU registers when there is a context switch between processes

- Each virtual address space corresponds to a ***contiguous range of physical addresses***

- The kernel is responsible for deciding *where* each virtual address space should map in physical memory

  - The OS must **track** which parts of PM are in use, and which parts are free
  - OS must allocate/deallocate variable-sized chunks of PM
  - this creates potential for ***fragmentation*** of physical memory

- What is the difference between external fragmentation and internal fragmentation?

Internal fragmentation is when there is wasted space within the allocated memory. For example a fixed size of memory is allocated to a process. but the amount allocated is bigger than needed. External fragmentation is wasted space between allocated memory, this is seen in the dynamic relocation example. The total free physical addresses space is greater than needed, but there is no contiguous range of physical addresses satisfies needed addresses.

Remember: 2^{12} **bytes = 4KB**

Paging: Physical Memory

Physical memory is divided into fixed-size chunks called **frames** or physical pages.

Paging: Virtual Memory

Virtual memories are divided into fixed-size chunks called **pages**. Page size is equal to frame size.

Paging: Address Translation

Each page maps to a different frame.

Address Translation in the MMU, Page Tables:

* The MMU includes **a page table base register** which points to the page table for the current process.

- Look up the PTE in the current process page table, using page number. If the PTE is NOT valid, raise an **exception**

- PTE may contain other fields(write protection bit used for read-only page, reference bit, dirty bit)

- A page table has one PTE for each page in the virtual memory
- page table size = #pages * size of PTE
- #pages = VMsize / PGsize

Page Tables are kernel data structure.

Paging reduces external fragmentation.

Kernel:

1. Manage MMU registers on address space switches(Context switch)
2. Creates & manage page tables
3. allocate/deallocate PM
3. Handle exceptions raised by the MMU

MMU(hardware):

1. Translate virtual addresses to physical addresses
2. Check for and raise exceptions when necessary

TLBs(Translation Lookaside Buffer):

Hardware-Managed TLBs:

- TLB is a small, fast, dedicated cache of address translations, in the MMU
- Each TLB entry stores a (page# -> frame#) mapping
- If the MMU cannot distinguish TLB entries from different address spaces, then the kernel must clear or invalidate the TLB on each context swtich from one process to another
- the MMU handles TLB misses, and must understand kernel's page table format

Software-Managed TLBs:

MIPS has a Software-Managed TLBs.

In case of a TLB miss, MIPS raise exception, the kernel must handle the exception and the instruction that caused the exception is re-tried.

- What is the difference between a hardware-controlled TLB and a software-controlled TLB? Which is the MIPS TLB?

Generally speaking, such hardware implementations are faster. However, the downside is that hardware dictates the form of the PTE. Which restricts what we can do in software.

The software managed TLB handles TLB misses by throwing exceptions and letting the kernel of the OS handle things. This gives us more **flexibility** in implementation, but, is generally slower.

Segmentation:

- Instead of mapping the entire virtual memory to physical, we can provide a separate mapping for **each segment of the virtual memory** that the application uses
- With K bits for the segment ID, we can have up to:
- 2^K segments
- 2^{V-K} bytes per segment
- Fragmentation of physical memory is **possible**
- Translating Segmented virtual addresses approach:

  1. MMU, similar as dynamic relocation

  2. segment table

Two-Level/Multi-Level Paging:

Directory table and page table

Post Midterm:

- Page tables for large address spaces may be very large. One problem is that they must be in memory,and must be physically contiguous. What is the solution to this problem?
- Multi-Level Paging

- What steps must the OS take to handle a page fault exception?
- The OS must determine which page table entry caused the fault, load the page into memory, set the valid bit, and then retry the instruction.

Virtual Memory for the Kernel:

Sharing problem: can be addressed by making the kernel's virtual memory **overlap** with process' virtual memories.

Boostrapping problem solution: architecture-specific

*** TLB for user address only, kernel address is translated by hardware**

* Attempt to access kernel code/data in user mode result in **memory protection exception**, **NOT** invalid address exceptions

User Space and Kernel Space on the MIPS R3000:

kseg0->physical address = addr - 0x8000 0000 // kernel code and data lives in kseg0

kseg1->physical address = addr - 0xA000 0000 // devices are accessed data and code through kseg1

Exploiting Secondary Storage:

Goals:

- Allow VM > PM
- Allow multiprogramming levels by using less of the available(primary) memory for each process.

Method:

- Allow pages from VM to be stored in secondary storage(disk,SSD).
- Swap pages(or segments) between secondary storage and primary memory, so they are in primary memory when they are needed

Resident Set and Present Bits:

- The set of virtual pages present **in physical** memory is called the **resident set** of a process

- A process's resident set will **change** over time as pages are swapped in and out of physical mem.

To track which pages are **in** PM, each PTE contains **present bit**.

- valid = 1, present = 1: page is valid and is in PM
- valid = 1, present = 0; page is valid but not in PM, need to be swapped in PM
- valid = 0, present = x: whatever x is(either 0 or 1), it is a invalid page

Page Faults:

- When a process tries to access a page that is not in PM(PTE: valid = 1, present = 0):

- on a machine with a **hardware-managed** TLB, the **MMU** detects this when it checks the pages's PTE(cuz MMU handles TLB miss), and **generates an exception**, which **kernel** must handle.
- on a machine with a **software-managed** TLB, the **kernel** detects the problem when it checks the page's PTE after a TLB miss(TLB miss => MIPS raise an exception => kernel handle the exception and check page's PTE)

- **Attempting to access a non-resident page** is called a **Page Fault.**

- What steps must the OS take to handle a page fault exception?

- When a page fault happens, it is the **kernel's job** to:

1. **Swap** the page into PM from secondary storage, **evicting** another page from PM if necessary(e.g. physical addr space is full).
2. **Update** the PTE(set the **present bit**)
3. **Return from the exception**, application retry the VM access which caused page fault

Secondary Storage is Slow!!!!

- access to disks: milliseconds(1,000 milliseconds = 1s)
- access to SSD: 10's-100's microseconds(1,000,000 microseconds = 1s)
- access to memory:100's nanoseconds(1,000,000,000 = 1s)
- Example: suppose secondary storage access is 1000 times slower than memory access:
    - 1 page fault every 10 memory access: 100 times larger than w/o swapping [(1*1000+1*10) / (10*1)]
    - 1 in 100, 10 times larger …. [(1*1000+100*1) / (100*1)]
    - 1 in 1000, 2 times larger [(1*1000+1000*1) / (1000*1)]

Performance with Swapping:

- To provide good performance for VM accesses, the **kernel** should try to ensure that **page faults are rare**.

- Some techs the kernel can use to improve performance:

- limit the number of processes, so that there is enough physical memory per process
- **try to be smart about which pages are kept in PM, and which are evicted**
- hide latencies, e.g. by **prefetching** pages before a process needs them

- What is prefetching and what is its goal? What are the hazards of prefetching? Which kind of locality does prefetching try and exploit?
    - prefetch: load pages into PM before a process needs them. goal is to hide latencies and improve performance. One of hazards is we don't know which pages a process will need them. Prefreching may try Spatial locality.

- Why is it better to suspend and swap processes than just to have them all run at once?
    - increase the number of page faults, take too much time

Replacement policies(kernel try to be smart):

1. **FIFO(first in first out)**
2. **MIN: optimal page replacement,** replace the page that will **NOT be referenced for the longest time**
    - requires knowledge of the **future**(almost impossible)

    To be continue...

- What are the two types of locality to consider when designing a page replacement policy?

Locality:

- Real programs do not access their VM randomly, Instead, they exhibit **locality**:
    - **temporal locality:** programs are more like to access pages that they have accessed recently
    - **spatial locality:** programs are likely to access parts of memory that are **close** to parts of memory they have accessed recently

Replacement policies continue:

3. **Least Recently Used(LRU) page replacement**

Measuring Memory Accesses:

- The kernel is **NOT** aware which pages a program is using **unless there is an exception**

- This makes it **difficult** for the kernel to exploit locality by implementing a replacement policy like LRU
- The **MMU** can **help** solve this problem by tracking page accesses in **hardware**

- Simple scheme: add a **use bit(or reference bit)** to each PTE. This bit:

- is set by the MMU each time the page is used(each translation on that page)
- can be read and cleared by the kernel

  This use bit provides a small amount of memory usage information that can be exploited by the kernel.

- Does the MIPS TLB include a use bit?
  - NO

Replacement policies continue:

4. **The Clock Replacement Algorithm**
   - one of simplest algo that exploits the use bit
   - identical to FIFIO, except that a page is “skipped” if the use bit is **set**
   - kernel set all use bits to 0 periodically

CODE :

```
while use bit of victim is set
    clear use bit of victim
    victim = (victim + 1) % num_frames
choose victim for replacement
victim = (victim + 1) % num_frames
```

---- after replacement is done, kernel return from exception, MMU retries translation, success=>set use bit to 1

Replacement policies(summary):

1. **FIFO**
2. **MIN: optimal page replacement,** replace the page that will **NOT** be referenced for the longest time(almost impossible)
3. **Least Recently Used(LRU) page replacement**(difficult to implement)
4. **The Clock Replacement Algorithm**

CPU scheduling:

Simple Scheduling Model:

- a set of **jobs** to schedule
- ONLY ONE job can run at a time
- Each job we are given, job arrival time(a_i), job run time(r_i)
- Each job we define:
    - response time: start to run - a_i(arrival)
    - turnaround time: finish time - a_i(arrival)
- decide some algo to achieve some goal, e,g min average turnaround time, or min average response time

Basic Non-Preemptive Schedulers:
FCFS: first come first serve, run jobs in arrival time order
- simple, **avoids starvation**
- pre-emptive variant: **Round-Robin**
    - has a quantum, run jobs in arrival time order, each job run quantum time and then preempt and go to the end of the ready queue. new arriving jobs go to the end of the ready queue

SJF: shortest job first - run jobs in increasing order of r_i
- **minimizes average turnaround time**
- long jobs may **starve**
- pre-emptive variant: **SRTF**(shortest remaining time first)
    - when new jobs come in, **compare** and choose the shortest remaining time job to run(preempt)

Basic Non-Preemptive Schedulers(summary):
- Non-preemptive Schedulers(only when the running thread gives up the processor through its own actions, i.e. Block, Exit, Yield):
    - FCFS
    - SJF
- Preemptive Schedulers(in addition to non-preemptive, force a running thread to stop running:
- Round-Robin(based on FCFS, add quantum)
- SRTF

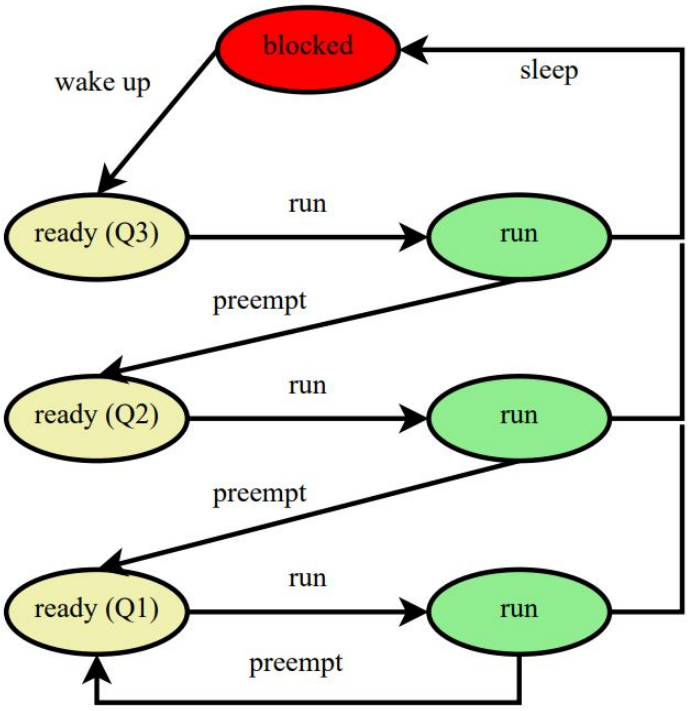
CPU scheduling:
- In CPU scheduling, “jobs” <-> threads
- CPU scheduling **typically differs** from the simple schedule mode(above)
    - the run times of threads are normally **UNKNOWN**
    - threads sometimes are not runnable: **BLOCKED**
    - threads may have **different priorities**
- The objective of the scheduler is normally to achieve a balance between:
    - responsiveness(ensure that threads get to run regularly)
    - fairness

- efficiency

Multilevel feedback queue:
- Objective: good performance for **interactive** threads, **non-interactive** threads make **as much progress as possible**
- Approach: given higher priority to interactive threads, so that they run whenever they are ready.
- Problem: how to determine which threads are interactive which are not??
potential answer: syscall - interactive threads, user program - non-interactive threads

Multi-level Feedback Queues(Algo):
- scheduler maintains n **round-robin <u>ready</u>** queues(Q1,....,Qn), Qn has highest priority, Q1 has lowest priority
- scheduler always chooses a thread from Qn **until** it is **empty**
    - if Qn is empty, choose a thread from Qn-1 until empty, and so on
- threads in queue Qi use quantum qi
    - typically larger quanta for lower-priority threads (qi >= qi+1) -> Interactive threads have higher priority, run whenever they are ready. Non-interactive threads have lower priority, make as much progress as possible
- if the running thread from Qi uses its entire quantum and gets preempted, **demote** it to queue Qi-1(go to the end of Qi-1)
- if a thread **blocks**, put it into **Qn when it wake up**
- to **prevent starvation**, **periodically move all threads to Qn**(so that CPU can run every threads once in shortest time because qn is normally the shortest quanta)

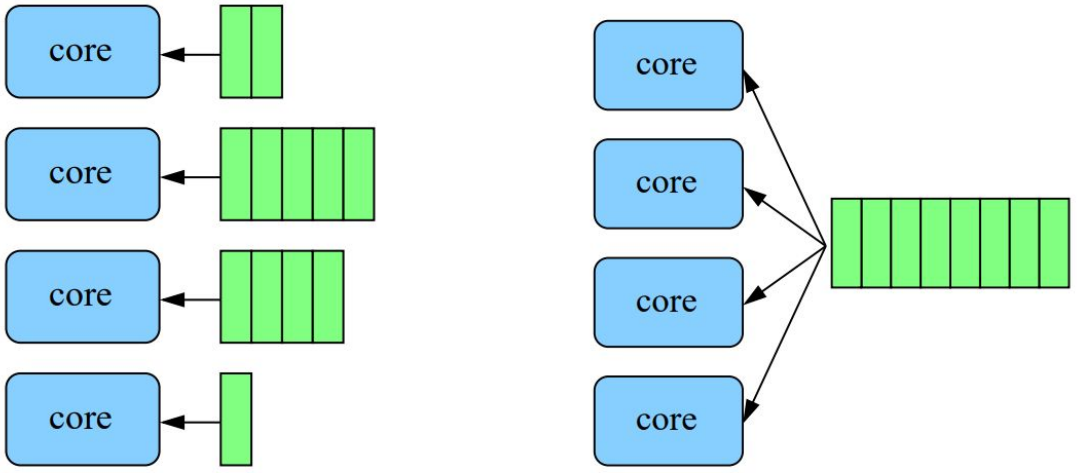
3 Level Feedback Queue State Diagram



Linux Completely Fair Scheduler(CFS) - main idea:
- each thread can be assigned a **weight w_i**
- goal is to ensure that each thread gets a “share” of the processor in **proportion** to its weight
- basic operation:
  - given actual runtime(AR) of i-th thread AR_i, calculate the virtual time(VR) of all threads and choose the thread with lowest VR to run.
  - $$VR_i = AR_i \cdot \frac{\sum_j w_j}{w_i}$$

Starvation? No

Scheduling on Multi-Core Processors



per core ready queue(s) vs. shared ready queue(s)

Need load balance Need Lock

Scalability and Cache Affinity(可扩展性和缓存亲和性):

- Contention and Scalability
  - access to **shared** ready queue is a critical section, **mutual exclusion needed(global queue lock in the kernel, when scheduler accesses shared ready queue, the lock is acquired, the scheduler assigns the core the front thread of the queue, then pop, then release lock)**
  - as number of cores grows, contention(竞争) for **shared** ready queue becomes a problem
  - **per-core design scales** to a larger number of cores
- CPU cache affinity
  - **as thread runs, data it accesses is loaded into CPU caches**
  - moving the thread to another core means data must be **reloaded** into that core's caches(each core has its own cache)
  - **as thread runs**, it acquires an **affinity** for **one core** because of the cached data
  - **per-core design benefits** from affinity by keeping threads on the same core
  - **shared queue design does NOT**

Load Balancing:

- in **per-core design**, queues may have different lengths, and this results in **load imbalance** across the cores
  - cores may be **idle** while others are busy
  - threads on lightly loaded cores get more CPU time than threads on heavily

loaded cores
- **Shared queue design doesn't** have this problem
- per-core designs typically need some mechanism for **thread migration to address load imbalances**

Shared queue design:
- advantage: no imbalance loading(Fair to all processes)
- disadvantage: not scalable
  Poor cache affinity

Per-core design:
opposite to Shared queue design

Device and I/O

Device Register:

Definition: A Device Register is the view any device presents to a programmer

- What are the registers each device maintains? How are they used?

- Device Register Types:
  - Status: tells you something about the device or its state, typically READ
  - Command: tell the device to do something -> WRITE
  - data: send/receive block of data to/from device R/W

Device Drivers:

Definition: a device driver is **a part of the kerne**l that interacts with a device

- To avoid polling, Device raise an exception when the status is "completed":

```
// only one writer at a time
P(output device write semaphore)
// trigger the write operation
write character to device data register
repeat {
    read writeIRQ register
} until status is ''completed''
// make the device ready again
write writeIRQ register to ack completion
V(output device write semaphore)
```

Polling: the kernel driver repeated checks device status

Device Driver Write Handler:

```
// only one writer at a time
P(output device write semaphore)
// trigger write operation
write character to device data register
```

Interrupt Handler for Serial Device:

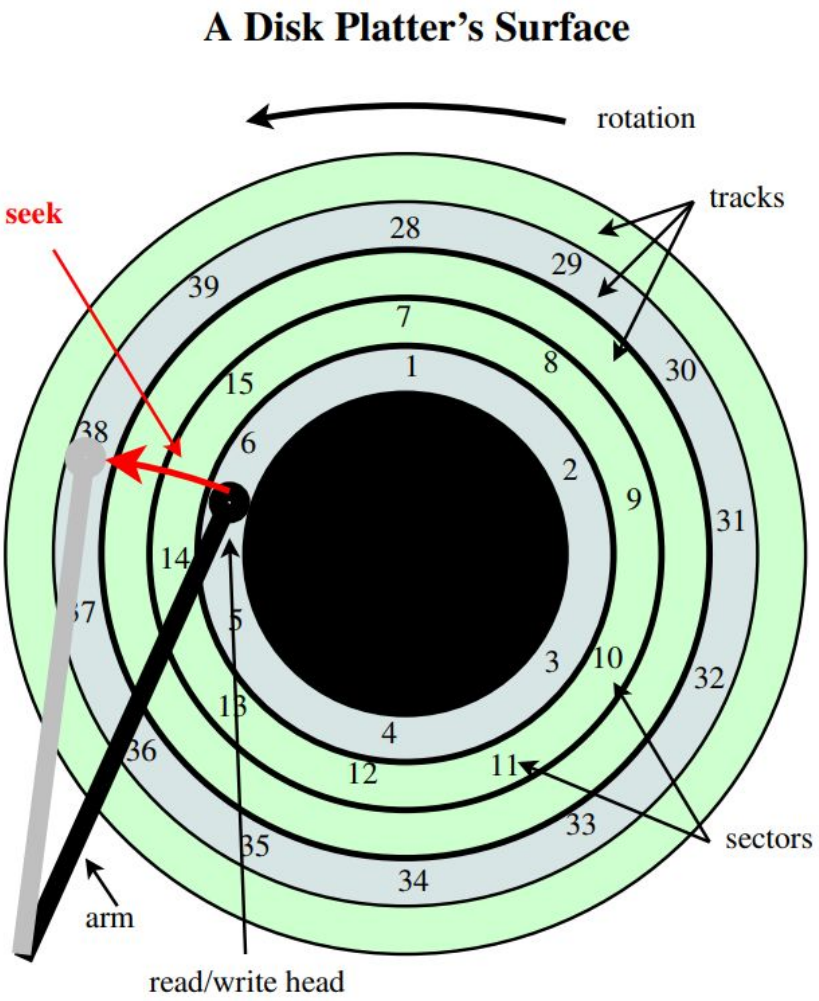
```
// make the device ready again
write writeIRQ register to ack completion
V(output device write semaphore)
```

Access Devices:

- The ways device driver access device registers:
    1. port-mapped I/O
        - device registers are assigned “port” numbers
        - instructions **transfer** data b/w **a specified port** and **a CPU register**
    2. memory-mapped I/O
        - each device register has a physical memory address
        - device driver can R/W to device registers using normal load and store instructions, as though accessing memory

Logical View of a Disk Drive:

- disk is an array of numbered blocks(or sectors)
- each block is the same size(e.g., 512 bytes)
- **blocks are the unit of transfer** b/w the disk and memory
    - typically, **one or more contiguous blocks** can be transferred in a single operation
- storage is **non-volatile**, i.e. data **persists** even when the device is **without power**



- each track holds same amount of data
- each track has same number of sectors
- each sectors has same amount of data

Cost Model for Disk I/O:

Seek Time: **move** the R/W heads to the appropriate cylinder

- seek distance = previous pos - current pos
- Best case: if next request is for data on the same track as the prev request
    - seek distance = 0, seek time = 0
- Worst case: seek from outermost to innermost track, max seek time
- CostToMove1Track = MaxSeekTime / #Tracks
- AvgSeekTime = MaxSeekTime / 2

Rotational Latency: wait until the desired **sectors spin** to the R/W heads

- depends on **rotational speed** of the disk
- e.g. a disk spins at 12000RPM, 1 complete rotation takes 5 millisecond(max rotational latency)
- CostToMove1Sector = MaxRotLatency / #SectorsPerTrack
- AvgRotLatency = MaxRotLatency / 2

Transfer Time: wait while the desired sectors spin **past** the R/W heads

- Transfer Time = #SectorsRW x CostTomove1Sector
- OR Transfer Time = (#SectorsRW / #SectorsPerTack) * MaxRotLatency

- Request service time = seek time + rotational latency + transfer time

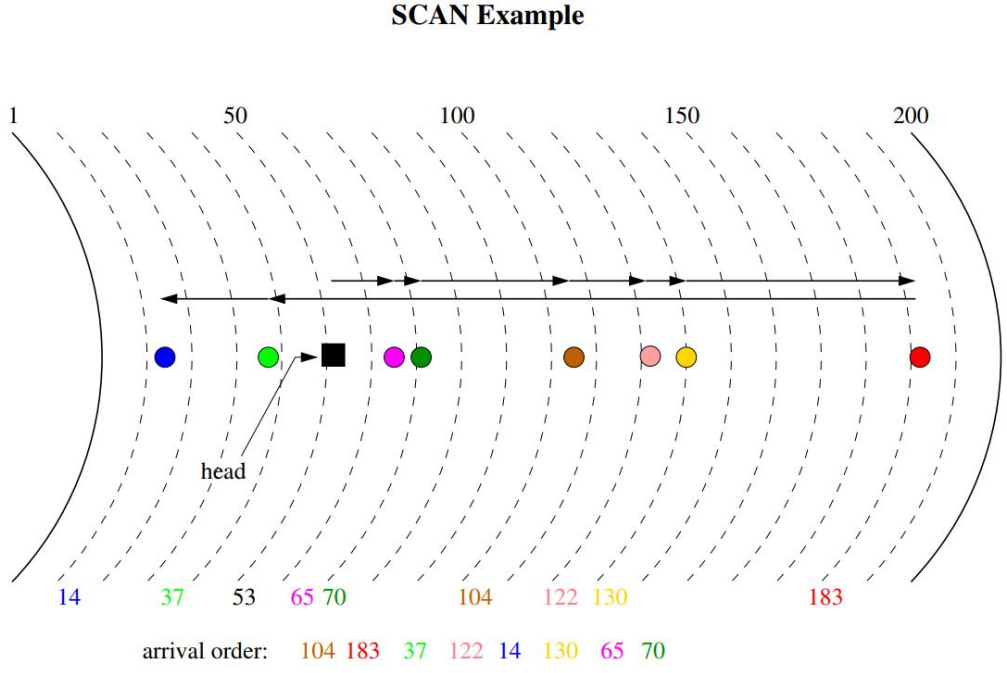
- **sequential** I/O is **faster** than non-sequential I/O(eliminate the need for most seeks)

Disk Head Scheduling:

- goal: **reduce** seek times by **controlling the order** in which requests are serviced
  - can be performed by either device or OS or both
  - there is a queue of outstanding disk requests to reorder
  - an on-line approach is required: new I/O requests may arrive at any time

Algos:

- FCFS: handle requests in arriving order, fair and simple, no starvation, no optimization
- SSTS(shortest seek time first): chooses closest request(greedy algo), **starvation**
- **Elevator Algo(Scan):** the disk head moves in **one direction until this are no more requests in front of it**, then reverse direction. **No starvation!**



Data Transfer To/From Devices:

- What is the difference between program controlled I/O and Direct Memory Access? What are their respective advantages? Which does SYS/161 use?

Option 1: program-controlled I/O

- The device driver moves the data b/w memory and a buffer on the device
    - Simple. but **CPU is busy** while the data is being transferred

Option 2: direct memory access(DMA)

- The device **itself** is **responsible** for moving data to/from memory. **CPU is NOT busy** during this data transfer(idle), and is free to do something else.

SYS161 uses program-controlled I/O

Writing to a Sys/161 Disk

Device Driver Write Handler:

```
// only one disk request at a time
P(disk semaphore)
copy data from memory to device transfer buffer
write target sector number to disk sector number register
write ``write'' command to disk status register
// wait for request to complete
P(disk completion semaphore)
V(disk semaphore)
```

Interrupt Handler for Disk Device

```
// make the device ready again
write disk status register to ack completion
V(disk completion semaphore)
```

Second Semaphore need?: This particular example, no, BUT... we do need it for the read ... and we would need that second semaphore if the write thread was to check for errors after the write had completed.

The disk semaphore prevents more than one operation at the disk at a time.

However, if we want the clock to beep, every time we push a key, then we DO NOT need the second semaphore. Why? Because it doesn't matter which thread instantiates the beep.

Reading From a Sys/161 Disk

Device Driver Read Handler:

```
// only one disk request at a time
P(disk semaphore)
write target sector number to disk sector number register
write ``read'' command to disk status register
// wait for request to complete
P(disk completion semaphore)
copy data from device transfer buffer to memory
V(disk semaphore)
```

Interrupt Handler for Disk Device

```
// make the device ready again
write disk status register to ack completion
V(disk completion semaphore)
```

We use a second semaphore whenever the originating thread MUST do something with the device or its registers prior to releasing the device. For example, when you read from a disk, the reading thread is the one that wanted the data from the disk --- so it should be the one that grabs the data from the device transfer buffer and stuffs it into its address space (where data would go).

Direct Memory Access(DMA):

- DMA is used for block data transfers b/w devices and memory
- Under DMA, the CPU initiates the data transfer and is notified when the transfer is finished. **However, the device(NOT CPU) controls the transfer itself.**

1. CPU issues DMA request to Device ---> Device receives
2. Device transfers data b/w memory and itself
3. Device notifies CPU(**interrupt**) the transfer is done ---> CPU receives, and it was idle during the data transfer

Solid State Drives(SSD):

- **NO mechanical parts**; use **integrated circuits** for persistent storage instead of magnetic surfaces
- DRAM: requires **constant** power to keep values
  - transistors with capacitors
  - capacitor holds **microsecond** charge; periodically refreshed by primary power
- Flash Memory: traps electron in quantum cage
  - floating gate transistor
  - usually NAND(not-and gate)

SSD Data Arrangement:
- logically divided into blocks and pages
    - 2,4, or 8KB pages
    - 32KB - 4MB blocks(each block contains several pages)
- reads/writes at page level
    - **pages are initialized to 1s**; can transition 1->0 at page level (**write** new page)
    - a high voltage is required to switch 0->1 (overwrite/ delete page)
    - **cannot** apply high voltage at page level, **only to blocks**, i.e. **overwrite/delete data commands must be done at block level**

Writing and Deleting from Flash Memory:
- Naive Solution(slow):
    - read whole block into memory
    - re-initialize block(all page bit back to 1s)
    - update block in memory, write back to SSD
- SSD controller handles requests(faster):
    - **mark** page to be deleted/overwriiten as invalid
    - **write to an unused page**
    - **update translation table**
    - requires garbage collection

Wear Leveling(耗损平均技术):
- SSD are not impervious, **blocks** have **limited** number of write cycles
    - if block is no longer writable, it becomes read-only
    - when a certain % of blocks are read-only; disk becomes read-only
- SSD controller wear-levels; ensuring that write cycles are **evenly** spread across all blocks

Defragmentation:
- Defragmentation takes files spread across multiple, non-sequential pages and makes them sequential
    - it **rewrites** many pages of **memory**, possibly several times
    - SSD random and sequential access have approximately the same cost
        - no clear advantage to defragmentation
        - extra, unnecessary writes performed by defrag - **causes pre-mature disk ageing**

File Systems:

Files and File Systems:

- files: persistent, named data objects
    - data consists of a **sequence** of numbered bytes
    - file may **change** size over time
    - file has associated meta-data
        - e.g. owner,access controls, file type, timestamps...
    - file system layer:
        - logical file system: open, close, seek
        - virtual file system(option): allow client applications to access different types of physical file system - Abstraction
        - Physical file system: NTFS, EXT4

File Interface: Basics:

- open:
    - open returns a file identifier(or handle or **descriptor**)
    - other operations (R/W) require file descriptor as a parameter
- close:
    - **kernel tracks** while file descriptors are currently **valid** for each process
    - close **invalidates** a valid file descriptor
- R/W, seek:
    - read copies data from a file into a virtual address space
    - write copies data from a virtual address space into a file
    - seek enables non-sequential R/W

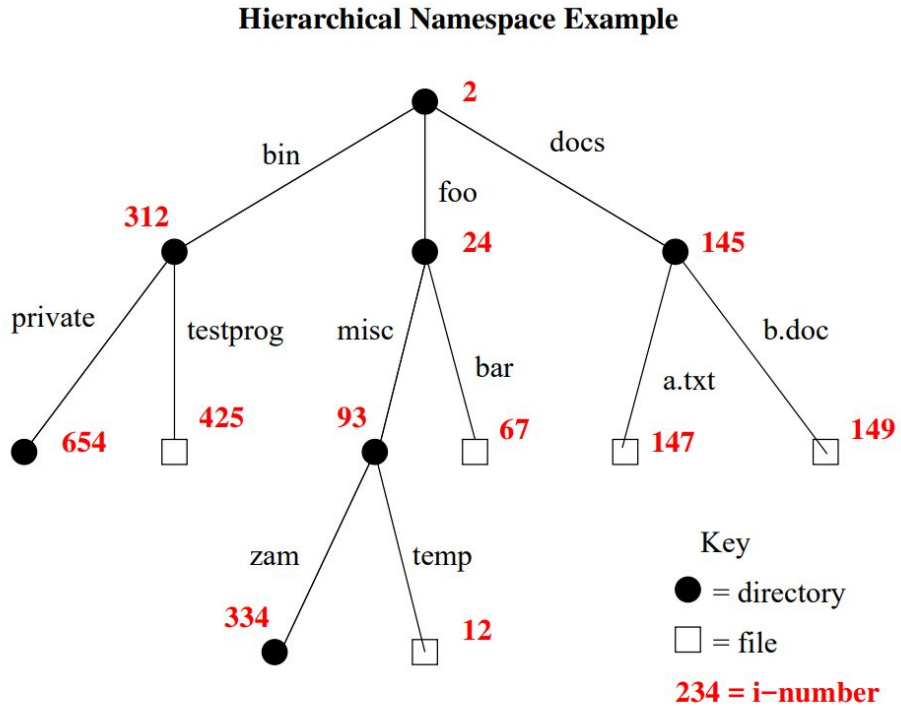
File Position and Seeks:

- each file **descriptor**(returned by open file) has an **associated file operation**
- R/W operations:
    - start from the current file position
    - **update** the current file position
- seeks are used for achieve non-sequential file I/O
    - *lseek* changes the file position associated with a descriptor, and do NOT actually read file
    - next R/W from that descriptor will use the new position

Directories and File Names:

- A **directory** maps **file name** (strings) to **i-numbers**
    - an i-number is a **unique**(within a file system) **identifier** for a file or directory
    - given an i-number, the file system can find the data and meta-data for the file
- **Directories** provide a way for applications to **group** related files
- In a directory tree, **files are leaves**

- **Applications** refer to files using **pathname**, NOT i-number



Links:

- A **hard link** is an association b/w a **name** and an **i-number**
    - each entry in a directory is a hard link
- When a file is created, so is a hard link to that file
- Once a file is created, **additional hard links** can be made to it
- Linking to an existing file creates a new pathname for that file
    - each file has a unique i-number, but may have multiple pathnames
- **NOT** possible to link to a directory (to avoid cycles)

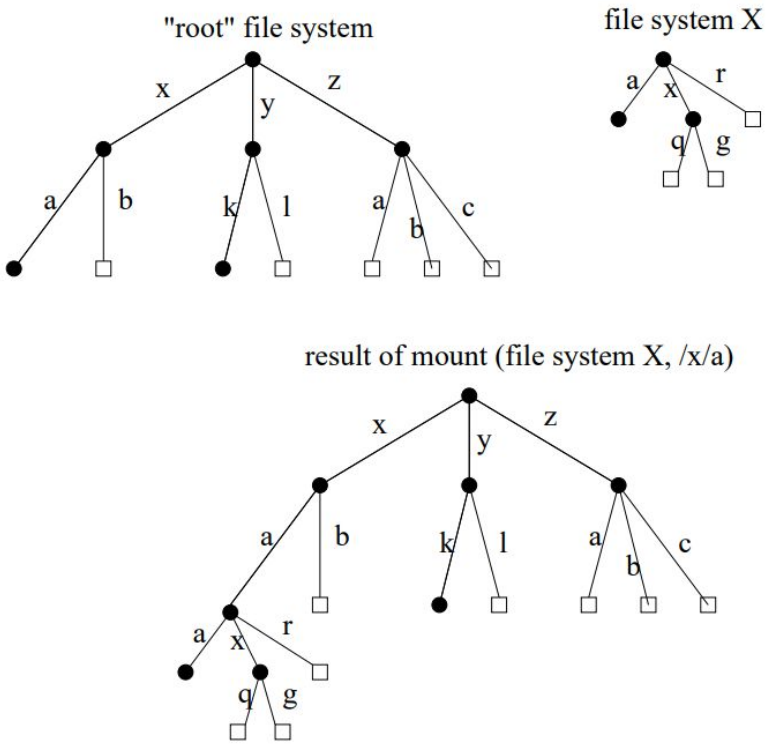
Unlinking:

- Hard links can be removed
- When the **last hard link** to a file is removed, the file is also removed

Multiple File Systems:

- It is common for a system to have multiple file systems, some kind of globale file namespace is required
    - DOS/Windows: use two-part file names: file system name(C:), pathname within file system
    - Unix: create single hierarchical namespace that combines the namespaces of two file systems - mount system
        - the new namespace is temporary - it exists only until the file system is unmounted

Unix `mount` Example

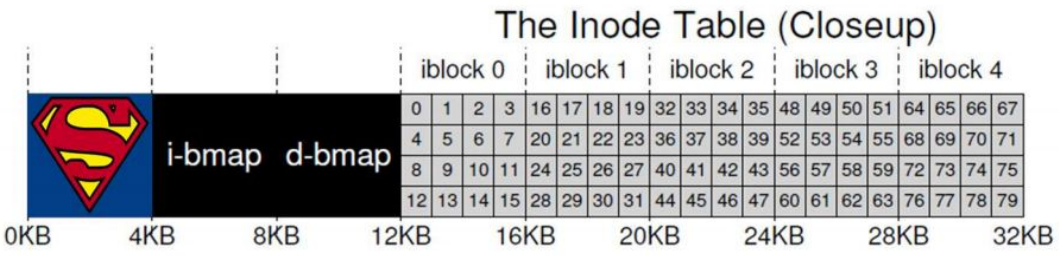


File System Implementation:

- Things need to be stored persistently
  - file data, file meta-data, directories and links, file system meta-data
- non-persistent info:
  - open files per process, file position for each open file, cached copies of persistent data

File System Example:

- **Memory** is usually **byte** addressable
- **Disk** is usually **sector** addressable



- Most of the blocks should be for storing user data(last 56 blocks)
  - two files cannot share one block

- Need some way to map files to data blocks
- Create an array of i-nodes, where each i-node contains the meta-data for a file
    - The index into the array is the file's index number(i-number)

- We also need to know which i-nodes and blocks are unused
- add an i-bitmap and data-bitmap, each bitmap uses one block

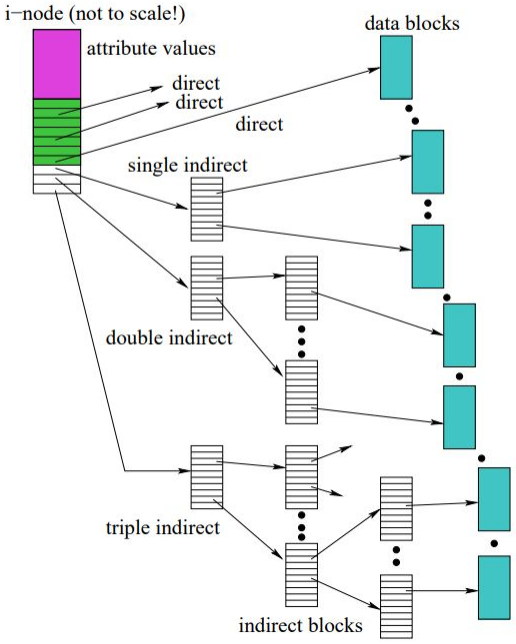
- Reserve the first block as the **superblock**
- A superblock contains meta-information about the entire file system
    - e.g. how many i-nodes and blocks are in the system, where the i-node table begins

i-nodes:
- An i-node is a **fixed size** index structure that holds both file meta-data and **a small number of pointers to data blocks**
- i-node may include: file type, file permission, file length, number of file blocks, time of last file access, time of last i-node update, number of hard links to this file, **direct data block pointers, single double and triple indirect data block pointers**

Direct pointer:
- each pointer points to a different block for storing user data
- Pointers are ordered

Indirect pointer: An indirect pointer points to a block full of direct pointers
- #PTRsPerBlock = BlockSZ / PTRSZ

i-node Diagram



Directories:
- Implemented as a **special** type of file
- Directory files contains directory entries, each consisting of
    - a file name(component of a path name) and the corresponding i-number
- Directory files permissions:
    - **Can be READ by application programs**
    - Are **only updated by the kernel**(e.g. create file, create link)
    - **Cannot be written directly by application programs**

In-Memory (Non-Persistent) Structures:
- per process
    - descriptor table
        - **open file descriptors**?
        - **to which file**?
        - **current file position?**
- System wide
    - open file table
        - currently open files(by any process)
    - i-node cache
        - in-memory copies of **recently-used** i-nodes
    - block cache
        - in-memory copies of **data blocks and indirect blocks**

Reading From a File(/foo/bar):

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

OPEN:
- First read the root i-node(i-node 2, i-node 1 is usually for tracking bacd blocks)
- Read the directory information from root
    - Find the i-number for foo in directory entry table

- Read the foo i-node

- Read the directory information form foo
  - Find the i-number for bar in directory entry table
  - Read the bar i-node(do permission check)
    - allocate a file descriptor in the per-process descriptor table
    - increment the counter for this i-number in global open file table(kernel)

Read:

- Find the block using a **direct/indirect** pointer and read the data
- update the i-node with a new access time(access time is only for files, not directory)
- update the file position in the descriptor table(process)
- closing a file deallocates the file descriptor in descriptor table and decrements the counter for this i-number in the globale open file table

Creating a File(/foo/bar):

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[1]
			read							
						read				
				read						
							read			
create (/foo/bar)		read write								
							write			
					read write					
				write						
					read					
write()	read write									
								write		
					write					
					read					
write()	read write									
									write	
					write					
					read					
write()	read write									
										write
					write					

Create:

- Read root inode
- Read root data to find i-number for foo
- Read foo inode
- Read foo data to prepare file position to write new directory entry(not allowed to write partial block)
- Read inode bitmap
- Write inode bitmap(create empty file)

- write foo data(directory entry)
- read bar inode
- write bar inode(modified time)
- write foo inode(modified time)

Write:

- Read bar inode
- read data bitmap
- write data bitmap
- write bar data(block level write, don't need to read)
- write bar inode

Problems Caused by Failure:

- a single logical file system operation may require several disk I/O operations
- deleting a file:
  - remove entry from directory
  - remove file index(inode) from inode table
  - mark file's data blocks free in free space index
- What if, because of a failure, some but not all of these changes are reflected on the dist?
  - system failure will destroy in-memory file system structures
  - persistent structures should be **crash consistent**, i.e. should be consistent when system restarts after a failure

Fault Tolerance:

- special-purpose consistency checkers:
  - runs after a crash, before normal operations resume
  - find and attempt to repair inconsistent file system data structure
    - file with no dir entry
    - free space that no marked as free

- journalling:
  - record operation log
  - after changes have been journaled, update the disk structures
  - after a failure, redo log operation