20-EECE-4029/6029, 20-ELTN-4022 - **Introduction to Operating Systems**

# Review Notes for Final Exam

December, 2013

## ACPI and Power Management

### Motivation

- Power consumption can be controlled. This is especially important for mobile devices.

### C States

- Power states
  - ▷ C0 - running
  - ▷ C1 - halt - main internal clocks off by software
  - ▷ C2 - stop - main internal clocks off by hardware
  - ▷ C3 - sleep - all internal clocks off
  - ▷ C4 - deeper sleep - clocks off, voltage reduced
  - ▷ C6 - deep power down - voltage reduced to anything

### P States

- Performance states
- A processor must be in state C0 to use a P state other than P0.
- Mainly control processor speed.
- Performance states allow OSPM to make tradeoffs between performance and energy conservation
- Performance states have the greatest impact when the states invoke different processor efficiency levels as opposed to a linear scaling of performance and energy consumption.

- Rough definitions are as follows:
  - ▷ P0 - maximum power and frequency
  - ▷ P1 - less power, lower frequency than P0
  - ...
  - ▷ Pn - n $\leq$ 16 - lowest power, lowest frequency

### Advanced Configuration & Power Interface

- ACPI allows the operating system to control the amount of power each device is given (e.g. standby or power-off).
- Also controls/checks thermal zones, battery levels, PCI IRQ routing, CPUs, NUMA (non-uniform memory access) domains and so on.
- Information about ACPI is stored in the BIOSs memory.
- Two main parts:
  1. Tables used by the OS for conguration during boot (# of CPUs, APIC details, NUMA memory ranges, etc).
  2. The run-time ACPI environment: AML code (a platform independent OOP language that comes from the BIOS and devices) and ACPI SMM (System Management Mode) code.

# Device Drivers

## Motivation

- Software that allows applications to interface easily with hardware devices

## Kernel module components

- Initialization
  - ▷ choose whether the device is memory mapped or not. If not, use `request_region` to have the OS allocate protected space. If so, use `request_mem_region` to reserve space and use `ioremap` to map the space in terms of pages
  - ▷ register the device with a "major" number. Use `register_chrdev`.
  - ▷ create a buffer. Use `__get_free_pages` to allocate it. When allocates using `kmalloc` the driver really does not own the pages, `kmalloc` owns the pages, the driver is just allowed to use the memory. This way the driver owns the space and can do what it wants with it
  - ▷ create a work queue for the interrupt handler bottom half
  - ▷ probe for a free interrupt line or validate one manually. See IRQ probing below.
- Cleanup
  - ▷ Disable the interrupt
  - ▷ Disable the tasklet or flush scheduled work
  - ▷ Unmap if memory mapped (use `iounmap`)
  - ▷ release the region obtained above
  - ▷ free the buffer space
- File operations - see Driver-11 as an example, including the above
  - ▷ read
  - ▷ write
  - ▷ poll
  - ▷ open
  - ▷ release

## Kernel considerations

- The kernel is limited to about 1GB of virtual and physical memory.
- The kernel's memory is not pageable.
- The kernel usually wants physically contiguous memory.
- Often, the kernel must allocate the memory without sleeping.
- Mistakes in the kernel have a much higher cost than they do elsewhere.
- Hence: kernel memory allocation is handled by `void *kmalloc(size_t size, int flags)` and `void kfree(const void *obj)` where 'flags' controls the behavior of memory allocation.

# Interrupts, barriers, timers

## Motivation

- interrupts are necessary due to slowness of I/O compared to other CPU tasks
- barriers are needed to prevent compiler optimizations from causing unpredictable results and to ensure that reads and writes are completed before the results of those operations are needed to be used
- timers may be used to recover from missed interrupts

## Interrupts

- There are numerous but finitely many interrupt lines. Some of the lines may be shared. See `/proc/interrupts` for interrupt counts and `/proc/stat` for a different view of the same information.
- When an interrupt is handled a "top half" is done very quickly to set up a "bottom half":
  - ▷ top half is called on the interrupt and is quick because interrupts are disabled to handle the event
  - ▷ top half sets up device data specific for the bottom half and schedules the bottom half
  - ▷ bottom half is where the heavy work is done and which may be put into a work queue or tasklet for processing while interrupts are turned on
- An interrupt can be missed - then what? Add a kernel timer that clears the state of the module and resumes normal operations after a certain amount of time
- Interrupts can be disabled. Often, interrupts must be blocked while holding a spinlock to avoid deadlocking the system.

## Interrupt handling

- **Tasklet**: tasklets are a deferral scheme that you can schedule for a registered function to run later. The top half (the interrupt handler) performs a small amount of work, and then schedules the tasklet to execute later at the bottom half.

  A given tasklet will run on only one CPU (the CPU on which the tasklet was scheduled), and the same tasklet will never run on more than one CPU of a given processor simultaneously. But different tasklets can run on different CPUs at the same time.
- **Work queue**: Work queues are a more recent deferral mechanism. Rather than providing a one-shot deferral scheme as is the case with tasklets, work queues are a generic deferral mechanism in which the handler function for the work queue can sleep (not possible in the tasklet model). Work queues can have higher latency than tasklets but include a richer API for work deferral. Deferral used to be managed by task queues through keventd but is now managed by kernel worker threads.

### Barriers:

- **Write memory barrier**: guarantees that all the write operations specified *before* the barrier will appear to happen before all the write operations specified *after* the barrier with respect to the other components of the system.
- **Read memory barrier**: guarantees that all the read operations specified *before* the barrier will appear to happen before all the read operations specified *after* the barrier with respect to the other components of the system.
- **General memory barrier**: guarantees that all the read and write operations specified *before* the barrier will appear to happen before all the read and write operations specified *after* the barrier with respect to the other components of the system.
- **Compiler barrier**: prevents the compiler from moving the memory accesses from one side of the barrier to the other side. This is a general barrier. The compiler barrier has no direct effect on the CPU, which may then reorder things however it wishes.
- **Lock**: memory operations issued after the lock *will* be completed after the lock operation has completed. Memory operations issued before the lock *may* be completed after the operation has completed. All lock operations issued before another lock operation *will* be completed before that lock operation.
- **Unlock**: memory operations issued before an unlock *will* be completed before the unlock operation has completed. Memory operations issued after an unlock *may* be completed before the unlock operation has completed. All lock operations issued before an unlock operation *will* be completed before the unlock operation. All unlock operations issued before a lock operation *will* be completed before the lock operation.
- **Note**: an unlock followed by an unconditional lock is equivalent to a full barrier, but a lock followed by an unlock is not.

## Deadlock, criteria, two-phase locking

See `http://gauss.ececs.uc.edu/Courses/c4029/code/deadlock/deadlock.html` for example code - (boy meets girl, boy falls for girl, girl doesn't)

### Four conditions to hold for deadlock to occur:

1. **Mutual exclusion**: threads claim exclusive control of resources that they require (e.g., a thread grabs a lock)
2. **Hold-and-wait**: threads hold resources allocated to them (e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks that they wish to acquire)
3. **No preemption**: resources (e.g., locks) cannot be forcibly removed from threads that are holding them
4. **Circular wait**: there exists a circular chain of threads such that each thread holds one more resources (e.g., locks) that are being requested by the next thread in the chain

### Prevention:

- **Circular wait**: provide total ordering on lock acquisition
- **Hold and wait**: acquire all locks at once, atomically
- **No preemption**: a trylock returns immediately if the lock cannot be acquired. But livelock is possible - threads are spinning but not getting anywhere
- **Mutual exclusion**: use structures that do not require locking - for example, make use of atomic instructions provided by the processor

### Avoidance:

- **Scheduling**: use global knowledge of which locks various threads might grab

### Detection and Avoidance:

- **Banker's algorithm**: process requests permission to use a resource. OS tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources. OS denies request if allocation requested is greater than the maximum for that resource, causes a wait if the number available is less than number requested. OS makes state check to test for possible deadlock conditions for all other pending activities. OS denies request if deadlock is possible. Otherwise, permission is granted.

### Detect and Recover:

- allow deadlocks to occasionally happen, then take action once a deadlock is detected. This may involve killing a thread or even restarting the operating system

### Two Phase Locking:

- **Guarantees serializability**: a transaction schedule is serializable if its resulting memory state is the same as the result of executing the transactions sequentially (that is, without overlapping in time). A transaction is a series of memory reads and writes.
- **Shared lock**: a transaction acquires a shared lock in order to read an item from memory. More than one transaction may acquire a shared lock but all transactions intending to write to a locked item must wait until the lock is released
- **Exclusive lock**: only one transaction can acquire an exclusive lock for writing to a memory item
- **Two phase protocol**:
  expanding: locks are acquired and none are released
  shrinking: locks are released and none are acquired
  end of phase 1: may be determined by some synchronization or atomic commitment point, but this is typically expensive and is often postponed until a final transaction is executed
- **Precedence graph**: a directed graph representing precedence of transactions in the schedule, as reflected by precedence of conflicting operations in the transactions
- a schedule is conflict-serializable if and only if its precedence graph of committed transactions is acyclic
- cycles of committed transactions can be prevented by aborting at least one uncommitted transaction on each cycle in the precedence graph of all the transactions

# Virtual Memory

## Motivation

- We want a process to think it has lots of space but we can't afford to give that much to it.
- So, we introduce the notion of virtual address space and a function that maps virtual addresses to physical addresses - physical space is allocated as needed.
- To support the sharing of libraries we introduce the Procedure Linkage Table and the Global Offset Table.
- We also segment the virtual address spaces to prevent alocating space that is not going to be used.
- But segmentation can be expensive because a translation function is required and because quite a bit of de-fragmentation may be necessary to satisfy requests for space.
- This last problem can be reduced by using fixed size segments called pages that are referenced through page tables.

## Demand paging

- Memory is allocated in pages, usually 4KB in size
- OS copies a disk page into memory *only* if an attempt is made to access it (a page fault was raised)
- Contiguous virtual space may not translate to contiguous physical space
- To get many more users in mainstore at once
- To get better utilization of hardware
- To handle jobs bigger than mainstore
- But it costs: performance & complexity
- Relies on: temporal locality - instructions in loops repeat often
- and spatial locality - instructions execute in sequence

## Components

- Caches, particularly in hardware, such as the Translation look-ahead buffer (TLB) for faster page table access
- Slabs, for kernel objects of known size
- Multi-level page tables: cache the cache of page table entries
- Page allocator: Buddy system
- Swapper: lazy

## Page fault

- If a page frame is not in memory a page fault is generated as follows:
  - ▷ an exception is raised - the OS handles it
  - ▷ the state of the process causing it is saved
  - ▷ it is determined that the exception was a page fault
  - ▷ it is determined that the address reference is valid
  - ▷ the location of the page on disk is found
  - ▷ the page is read from disk - this requires a wait to complete a seek plus some latency
  - ▷ the process that raised the exception is restarted because the CPU dropped it to act on another
  - ▷ the state of current I/O process is saved
  - ▷ the page table is updated
  - ▷ the state of process causing the fault is restored
  - ▷ the process is resumed
- Service time can be as high as 30 msec
- Memory access time is about 60 nsec - 6 orders of magnitude difference!!!
- For 10% impact on access speed we can can afford only 1 swap for every 2 million accesses
- If there is no free frame for the request, some extra tasks are added:
  - ▷ a "victim" page frame is found and written to disk (if it is modified)
  - ▷ the free-frame table is updated
  - ▷ the new page is read into the frame
  - ▷ the page table is updated
  - ▷ the process is restarted and resumed

## Page replacement

- If a page must be swapped out to make room for a new frame, a victim frame must be chosen and swapped out. The following are strategies for choosing a victim:

  ▷ **FIFO**: remove the frame that has been resident the longest subject to an anomaly: increased frames may cause more faults!

  ▷ **OPT**: remove the frame that will be next accessed the furthest into the future. This cannot be achieved in any practical sense although it is OK for some applications. It is primarily used as a benchmark to test other strategies against

  ▷ **LRU**: remove a frame that has not been used for the longest period. LRU may be implemented using a stack: when a frame is accessed it is pulled from the stack and placed on top, the victim frame is the one on the bottom. But this is hard to implement efficiently so approximations are used

  ▷ **LRU***: approximate LRU with a single reference bit in the page table entry. The victim is the first page with a 0 reference bit. Perhaps 0 out reference bits every so often so a frequently accessed frame will always have its reference bit set and it will be less likely to be swapped out

  ▷ **LRU+**: the *second chance* algorithm. All pages are in a circular queue. If the cursor points to a page with reference bit set then reset it (giving it a second chance) and move the cursor to the next page to test again. If all reference bits are set the one the cursor pointed to originally is the victim. This is easily implemented in software

  ▷ **LRU#**: expand on LRU* by adding a dirty bit. Define four classes of page:

    1. dirty/ref = 0,0: pages of this class can be swapped out easily because it is not likely that a reference will be made soon and it is not necessary to write the page to disk
    2. dirty/ref = 0,1: is similar except a write is necessary
    3. dirty/ref = 1,0: probably should not be swapped out because a reference is likely
    4. dirty/ref = 1,1: certainly is last to be considered for swap

## Frame allocation

- We should also be aware that every process should be granted some minimum number of page frames or else it will be competing with itself for free pages

- **Fixed Allocation**: all processes get the same minimum number of free pages Some processes get starved, some are too wealthy

- **Proportional Allocation**: processes get a number of pages that is proportional to either their total size or a mix of size and priority

- **Global**: a victim frame may be selected from any frame, allowing one process to "steal" from another. Performance is on a single process is non-deterministic (unpredictable)

- **Local**: a victim frame may be selected only from the set belonging to the evicting process. Results in more predictable performance but throughput suffers

### Thrashing:

- system condition where processes are spending more time on page faults than on useful work
- cause: processes have not been given enough frames there are too many processes running
- cure: based on locality
  - ▷ define *working-set-window* (WSW) as some number of instructions and define *working-set* as all pages accessed within the last WSW. If the sum of all WS sizes is greater than number of pages available then thrashing results. In that case, suspend some jobs as needed to keep the sum less. A working set changes slowly with time. A crude way to keep up with changes is to look at a few bits that are updated at regular intervals and remove pages from the working set whose bits have fallen off the table
  - ▷ decide on minimum and maximum allowable page fault rates per process. If a rate falls below the minimum, remove allocated pages from the process. If the rate rises above the maximum, add free pages to the working set
- control: page size
  - ▷ if page size is increased there will be fewer page faults which reduces time overhead spent waiting for I/O to disk. But internal fragmentation will be increased. However, smaller page size improves locality (matches locality of the process) which reduces I/O to disk. A larger page size means smaller page tables.

### Performance enhancers:

- **I/O interlock (locked pages)**: for code required for I/O fault handling, buffers, partially updated pages, critical kernel code, performance critical data. If, for example, an I/O process is given an address from which to take or put data, it must make sure the data at that address does not change for the long period of time that the I/O process takes - note the process should itself be interruptible.
- **Inverted page table**: a single page table replaces one page table per process. Implemented as a hash table, indexed on process and page number. These can be a lot smaller than a collection of many page tables.
- **Demand segmentation**: old processors do not have hardware support for demand paging and use demand segmentation instead
- **TLB reach**: the total memory that the TLB can cover is (TLB size)*(page size). If the reach is smaller than the working set size page faults increase
  - ▷ solution 1: increase TLB size. But that could go out of control
  - ▷ solution 2: increase page size. But that has problems noted above
  - ▷ solution 3: allow many different page sizes. But TLBs currently are designed to accomodate one size so this has to be worked out with chip manufacturers. Perhaps this is why linux only has sizes 4K and 4M - the latter maintained in software

## Performance improvements:

- **Copy-on-write**: if multiple processes request resources which are *initially indistinguishable* they can all be given pointers to the same resource. Then if a process tries to modify its "copy" of the resource, a separate (private) copy is made for that process to prevent its changes from becoming visible to all others. If no process ever makes any modifications, no private copy need ever be created.
  - ▷ **Main use**: when a process creates a copy of itself, the pages in memory that might be modified by either the process or its copy are marked copy-on-write. When one process modifies the memory, the kernel intercepts the operation and copies the memory so that changes in one process's memory are not visible to the other.
  - ▷ **Other use**: calloc returns 0'ed memory. Initial calls return pointers to the same page. Then, as writes are made, copies are made and pointers changed. This is done for large callocs
  - ▷ **Implementation**: the MMU is notified that certain pages in the process's address space are read-only. When data is written to these pages, the MMU raises an exception and the handler allocates new space in physical memory and makes the page being written correspond to that new location in physical memory.
  - ▷ **Advantage**: the ability to use memory sparsely. Usage of physical memory only increases as data is stored in it. Efficient hash tables can be implemented which only use little more physical memory than is necessary to store the objects they contain.
  - ▷ **Problem**: such programs run the risk of running out of virtual address space. Virtual pages unused by the hash table cannot be used by other parts of the program.
  - ▷ **Problem**: complexity. When the kernel writes to pages, it must copy any such pages marked copy-on-write.
- **Memory Mapped Files**: a resource is read into memory using demand paging. Reads and write are treated as ordinary memory reads and writes. Speeds up access to the resource. Allows several processes to share access to the resource.
  - ▷ **Problem**: a 5KB resource maps to two 4KB pages, so there is some waste
  - ▷ **Problem**: when a block of data is loaded in page cache, but is not yet mapped into the process's virtual memory space, page faults may occur
  - ▷ **Problem**: a file larger than the addressable space can have only portions mapped at a time, complicating reading it. An IOMMU can remedy this situation.
  - ▷ **Advantage**: a system call takes orders of magnitude longer than a memory access (e.g. seek time and latency are eliminated)
  - ▷ **Advantage**: the mapping can be to the kernel's page cache and therefore not take away from user pages

# Input-Output

## Motivation

- Since devices are typically slow compared to CPU clock speeds, they connect to busses other than the memory bus. Examples of such buses are Universal System Bus (USB), Peripheral Component Interconnect (PCI), Industry Standard Architecture (ISA), AT Attachment (ATA), Serial AT Attachment (SATA), Integrated Drive Electronics (ISE), Small Computer System Interface (SCSI).

## IO Channel

- The wire arrangement of the bus plus and the protocol it supports
- A device controller executes commands from a computer's host controller to cause desired actions on a device. Those commands as well as data and status to and from devices are sent over an IO channel
- Associated with devices are logical bus addresses which are used to access registers on device controllers
- Devices may also have data areas covering a large range of physical addresses

## Memory Mapped IO

- An alternative to IO channels is memory-mapped IO where the memory and registers of the IO devices are mapped to RAM addresses.
- When a CPU accesses what appears to be a RAM address, it is actually addressing memory of the device.
- The same instructions used for accessing RAM are used to access device memory in this case. It follows that such address ranges must be reserved for IO only, even if temporarily. See DMA below.

## Polling (not on)

- Mechanism to initiate an IO transfer
- A host controller may poll a status register of a device at regular intervals to determine whether some action needs to be taken or has completed
- A poll cycle may look like this:
  - The host reads the busy bit of the device controller until it's clear
  - The host writes a command in the command register and writes a byte into the data-out register
  - The host sets the command-ready bit
  - The device controller sees the command-ready bit is set and sets the busy bit
  - The device controller reads the command register
  - The device controller performs I/O operations on the device
  - If the read bit is set instead of write bit, data from the device is loaded into the data-in register, which is read by the host
  - The device controller clears the command-ready, error, and busy bit
- Polling may be OK for transfers that are known to happen in rapid succession for example sending bytes to a printer.

### Interrupts

- A device may connect direct to an interrupt request line (IRQ) on the processor to generate an interrupt or
- while executing a function, the code jumps to an address which is the entry point of a function for handling an interrupt
- An interrupt-driven IO cycle might look like this:
    - A device driver (kernel) issues a ”read” to the device controller
    - The device driver sleeps - the CPU performs other tasks
    - The controller reads data into a buffer and raises an interrupt request
    - The CPU reads the request and invokes the driver’s interrupt handler
    - The interrupt handler reads the buffer and processes it

### Direct Memory Access (DMA) (no detail is on)

- Direct Memory Access is a hardware mechanism that allows peripherals to transfer their IO data directly to and from main memory without the need to involve the system processor. The transfer is directly between the IO device and the memory.
- Direct Memory Access is important because it can greatly increase throughput to and from a device, eliminating significant computational overhead
- Three matters of concern with DMA are
    - Hardware support is required - but this is routinely supplied now
    - DMA “steals” cycles from the processor to make the transfer - so too frequent transfers may slow things down
    - Synchronization mechanisms must be provided to avoid accessing non-updated information from RAM - cache coherency must be preserved
- A DMA transfer cycle may look like this if initiated in software
    1. Device driver allocates a DMA buffer, sends signal to device indicating where to send the data, sleeps
    2. Device writes data to DMA buffer, raises interrupt when finished
    3. Device driver interrupt handler gets data from DMA buffer, acknowledges interrupt, awakens software requesting data to process it
- A DMA transfer cycle may look like this if initiated from the device
    1. Hardware raises an interrupt to announce that new data has arrived
    2. Interrupt handler allocates a buffer, tells the hardware where to transfer the data
    3. Device writes the data to the buffer, raises another interrupt when transfer is done
    4. Interrupt handler dispatches the data, awakens any relevant process, and takes care of housekeeping

- A DMA transfer cycle may look like this if initiated with a network device (uses a ring buffer)
  1. An incoming packet is placed in the next available buffer in the ring
  2. An interrupt is raised by the device
  3. The device driver sends the to packet to kernel code that will process it
  4. The device driver inserts a "fresh" buffer into the ring (one whose contents has been processed - all buffers are allocated once, during initialization)
- DMA transfers are over contiguous memory due to PCI constraints, PCI addresses are physical, and DMA controller addresses are logical bus addresses
- There needs to be some translation from one to the other
- To ensure the buffer occupies contiguous space:
  1. Use the DMA zone (lower 16MB of RAM) - works for some devices
  2. Allocate upper memory at boot, use portions of that space for buffers
  3. Use scatter/gather IO if possible
- Bounce buffers are created when a driver attempts to perform DMA on an address that is not reachable by the peripheral devicea high-memory address, for example. Data is then copied to and from the bounce buffer as needed. This could cause transfer slowdowns.

**Scatter/Gather IO** (not on but you benefit from knowing)

- Scatter/Gather IO is a method of input and output by which a single procedure call sequentially writes data from multiple buffers to a single data stream or reads data from a data stream to multiple buffers. The buffers are given in an array of sg objects.
- Scatter/gather is particularly effective in network IO (streams)

**Cache Coherency** (important)

- Cache coherency is a state whereby duplicated data elements have the expected value when they are accessed. Speed is generally improved by copying data (say from memory) to (fast) caches and operating on the caches. Modifications may cause a mismatch between the cached and memory-resident data. Mechanisms are needed to ensure that all copies are in sync when a read or write occurs.

**Block Devices**

- A block device is a data storage device that supports reading and writing data in fixed-size blocks, sectors, or clusters. Examples are: hard drives, optical disk drives, USB flash drives
- A block device generally requires few pins so is in a compact package and a block device interface is easy to write.
- But a block device based on a given solid-state memory may be slower than a character device based on the same kind of memory because:
  - *read*: begins with a seek to the start of the block, the whole block is read including unneeded data
  - *write*: begins with a seek to the start of the block, the whole block is read into memory, data is modified, seeks to the start of the block, writes the whole block to the device.

### Character Devices

- A character device supports reading and writing data one character at a time, in a stream. Examples are: keyboard, sound card, network card, mice.

### Blocking IO

- Synchronous - a process is suspended until IO is completed
- Easy to use and understand
- Insufficient for some needs, required by others
- Reduces performance and throughput, unless required
- Can use multi-threading to improve or simulate asynchronous I/O

### Non-blocking IO

- IO call returns as much information as is available
- User interface, data copy (buffered I/O)
- Implemented via multi-threading
- Returns quickly with count of bytes read or written

### Asynchronous IO

- A process runs while IO executes
- Sometimes can be difficult to use correctly and well
- IO subsystem signals process when IO completed

### IO Protection

- User process may accidentally or intentionally attempt to disrupt normal operation via illegal I/O instructions. Therefore
  1. All IO instructions must be run in privileged mode
  2. IO must be performed via system calls
  3. Memory-mapped and IO port memory locations must be protected too
- This means that all I/O must be done by OS

### Streams

- A stream is a full-duplex communication channel between a userlevel process and a device driver
- A stream consists of:
  - a head end to interface with the user process
  - a driver end to interface with the device
  - zero or more stream modules between them
- Each stream module contains a read queue and a write queue Message passing is used to communicate between queues
- Stream IO is asynchronous, except when the user process communicates with the stream head

## IO Performance

- IO performance may be improved by doing one or more of the following:
  - Reduce number of context switches
  - Reduce data copying (e.g. use scatter/gather)
  - Reduce interrupts by using large transfers, smart controllers, polling
  - Use DMA to increase concurrency
  - Balance CPU, memory, bus, and IO performance for highest throughput
  - Reduce number of opens/closes, setup/teardown, etc.

# File Systems

## Motivation

- It is hard to imagine an operating system without file system support

## Typical file operations

- create - find space and add an entry to the directory
- write data at current file position pointer location and update pointer
- read file contents at pointer location, update pointer
- reposition the cursor within the file (seek)  change cursor location
- delete - free space and remove entry from directory
- truncate  delete data starting at cursor

## Access methods

Files occupy some number of blocks of space on the medium in which they reside. Those blocks may or may not be contiguous. Every byte has a location in the file: the offset in bytes from the beginning of the file. Every byte has a location on its medium. If a file is on a hard drive a location is specified by a cyclinder, sector, and offset.

- Sequential - information is processed in order. Sequential reads are easy as long as the location of the next block is known.
- Direct (aka Relative) - a file consists of fixed size records, indexed in some given order. Some algorithm must be applied to locate a record if the blocks are not contiguous in the medium containing the file (which is usually the case).
- Indexed sequential - a small master index points to disk blocks of a secondary index the records of which point to the actual file records. The file is kept sorted on a defined key. A binary search of the master index provides the block number of the secondary index and is used to locate a particular item.

## Directory structure

A flat directory structure may be fine if the number of files in a file system is very small but in current use flat structures are extraordinarily inefficient. All file systems have a root (top) directory.

- Tree - directories contain files and other directories. A file system object (file or directory) contained in a directory is a descendant of that directory. A directory containing a file object is an ancestor of that object. If `A` is an ancestor of `B` and `B` is an ancestor of `C` then `A` is an ancestor of `C` (transitive property). The descendant relation is also transitive. If `A` is an ancestor of `B` there is a sequence `A`, $\mathtt{x}_1$, $\mathtt{x}_2$, ..., $\mathtt{x}_N$, `B` such that `A` is an ancestor of $\mathtt{x}_1$, $\mathtt{x}_N$ is an ancestor of `B` and for all $1 \leq i < N$, $\mathtt{x}_i$ is an ancestor of $\mathtt{x}_{i+1}$. If `A` is the root directory then this sequence is a *path* to `B` and is written $\mathtt{/A/x_1/x_2/.../x_N/B}$. In a tree structure no file object has two different ancestors and the path to a file object is unique. A tree structure supports searched more efficiently because they typically originate in a directory that is many ancestors removed from the root. Listing the contents of a directory is usually quick because a directory usually contains a small number of file system objects - it is also easier for a human to understand the contents of a directory if it is properly organized.

- DAG - there may be more than one path to a file system object but no path contains the same file system object more than once. A DAG structure may be achieved through hard and soft links (see inodes notes below for the difference). With DAGs it is possible to share directories between users. It is also possible to alias files (this could be good or bad).
- Graph - any type of path is allowed including one that contains a file system object more than once. This could present problems which searching as search is inherently a recursive process. However, by marking visited directories, a backtrack mechanism can easily be implemented that avoids visiting the same directory twice - an example is known as depth-first search.

### Network file systems

- Allow sharing of files across computers and across users.
- Protocols to support this type of file system are designed based on balancing the following considerations:
  1. Simplicity and extendibility
  2. Efficiency - low overhead, low latency due to the protocol
  3. Failure recovery - many types of failures to worry about compared to file system manipulation on one computer. For example, how does the file system recover after a network interruption?
  4. Consistency semantics - how users are to access shared files. For example, should changes due to writes to an open file be visible immediately to all users who are also looking at that file? Should the protocol support *immutable shared files*?

### File system implementation

- Operating system provides an interface for the applications programmer (e.g. open, close, seek, etc.)
- A device driver is used for the bulk transfer of data - typically one or more logical blocks at a time
- A volume control block is contained in a file system and provides information about it. For example, in linux ext4 file system the control block contains: volume name, directory last mounted on, ID, magic number, revision #, features, flags, default mount options, current state, errors behavior, inode count, block count, reserved block count, free blocks, free inodes, first block, block size, fragment size, reserved GDT blocks, blocks per group, gragments per group, inodes per group, inode blocks per group, flex block group size, time created, last mount time, last write time, mount count, max mount count, date last checked, check interval, lifetime writes (mine has 1369 GB!), reserved blocks uid, reserved blocks gid, first inode, inode size, required extra isize, desired extra isize, journal inode, first orphan inode, default directory hash, directory hash seed, journal backup, journal features, journal size, journal length, journal sequence, journal start.
- Some of the space used by a file system is allocated for file-specific information such as creation time, time last modified, one or more pointers to blocks that contain pieces of the file. This is called a file control block.
- A file system may have a boot control block which contains information needed to boot the operating system

## File system layers

Layering the file system support code is useful for reducing complexity and redundancy, but it adds overhead and can decrease performance. It also enables the easy introduction of new file systems and types. For example, in addition to the normal, journaled file system for hard drives an OS may support a network file system, or a file system that automatically encrypts and decrypts when reads and writes are executed. The proc file system is a rather bizarre one compared to conventional file systems in that it enables a user to directly control kernel functions and check the status of the kernel.

- Device - hardware such as hard drive, flash drive, network card, and so on
- IO Control - device drivers provide the link between the hardware and the software. The operating system performs many difficult operations under the hood to facilitate portable code that may be useful for many applications.
- Basic file system - routines that perform the difficult operations mentioned above. A simple example is the translation of a logical address to a physical hard drive location. This function may partially be implemented in the device controller.
- File organization - takes care of higher-level aspects of a file system such as free space management and disk allocation.
- Logical file system - the API presented to the applications programmer and manages access control.

## File system organization: FAT

The FAT (file allocation table) file system format was used in TOS (Atari ST), MSDOS and even Windows 9X in the 1980s and into the 1990s. It is still used for USB flash drives because it is understood by just about all operating systems.

- Boot sector (0x0000 - 0x01FF) - Jump instruction, OEM name, BIOS parameter block, Signature
- Directory table - a file that corresponds to the contents of a particular directory, each being described in a 32 byte directory entry.
- Directory entry - 14 bytes for a name and attributes, 2 bytes each for create time, create date, last access date, file access rights, last modified time, last modified date, 2 bytes for first cluster occupied by the file (see FAT), 4 bytes for the size of the file in bytes. An example entry for a regular non-executable file named `HOWDY` of 15999 bytes (`0x3E7F`) beginning at cluster `0x42` allowing owner to read/write but all others to just read, excluding the time and date fields which are shown dotted, is the following:

| HOWDY | 0x00 | .. | .. | .. | 0x0AAD | .. | .. | 0x0042 | 0x0003E7F |
|---|---|---|---|---|---|---|---|---|---|

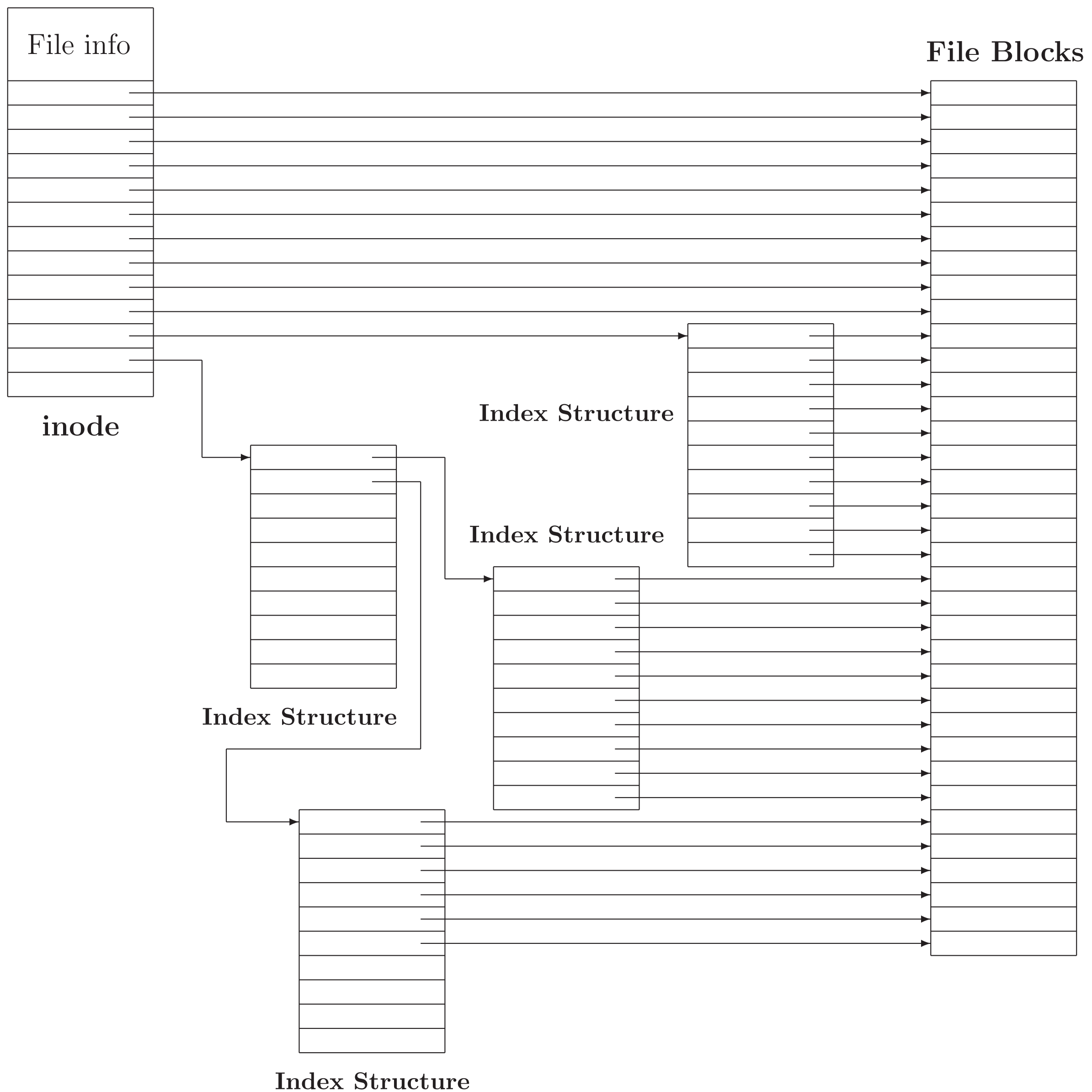
- FAT - usually two of these are present. A FAT contains a list of entries that map to each cluster (4 sectors or 2KB) on the volume. Each entry records one of the following: the cluster number of the next cluster in a linked list of cluster numbers indicating file position; a special end of cluster-chain (EOC) entry that indicates the end of the linked list; a special entry to mark a bad cluster; a zero to note that the cluster is unused. Each entry in a FAT corresponds to a cluster in the filesystem. Thus a linked list corresponding to a file (say `HOWDY`) whose data is found in clusters `0x42`, `0x44`, `0x46`, `0x47` (in that order) might look like this where entry `0x44` represents cluster `0x42`:

| ... | 0x44 | 0x81 | 0x46 | 0x05 | 0x47 | 0x00 | 0x67 | ... |
|---|---|---|---|---|---|---|---|---|

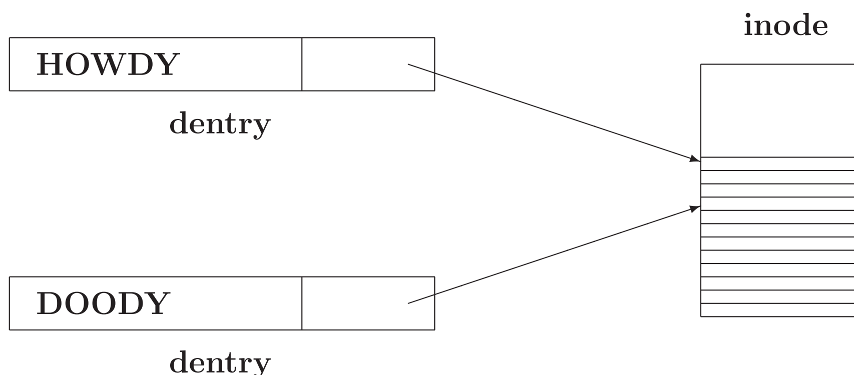
## File system organization: Unix

Linux uses inodes, superblocks and dentry objects to support file system manipulations

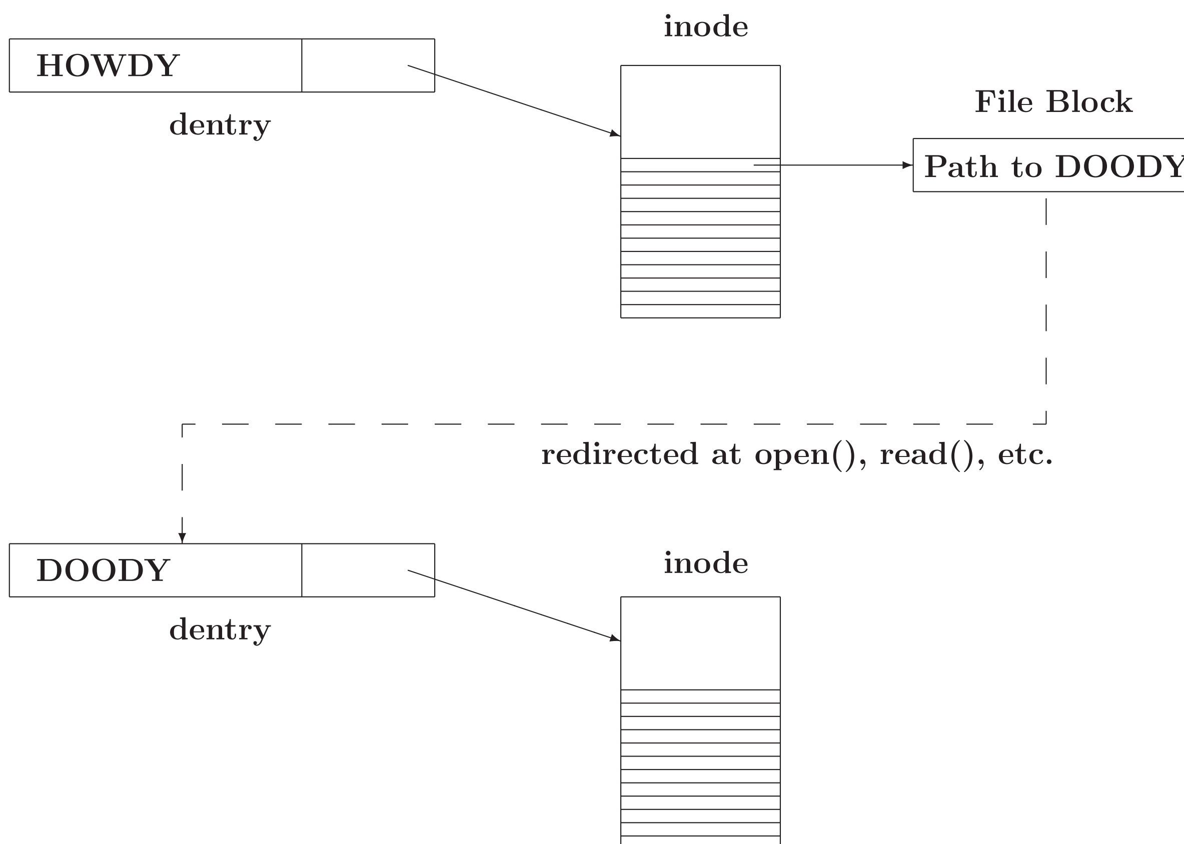
- inode - data structure that represents a file or directory. Includes data about the size, permission, ownership, and location on disk of the file or directory. For discussion, an inode holds a series of 13 pointers from which all blocks of a file may be accessed (several inode variants use different numbers of pointers and the exam will state the numbers to use should this be examined). The first 10 pointers directly indicate a block which contains file data. The 11th pointer references an index structure with 10 pointers that directly reference the next 10 blocks of the file's data. The 12th pointer references an index structure with 10 pointers each pointing to an index structure with 10 pointers that directly reference a file's data blocks. The 13th pointer references an index structure with 10 pointers referencing 10 index structures each with 10 pointers referencing index structures, each with 10 pointers referencing a file's data blocks. This is illustrated in the diagram below for a file occupying 36 blocks. Thus, whereas the there is a linear search required to locate a byte in a FAT file system, the search required with an inode is linear.



- superblock - this is the volume control block for unix and its contents is shown in the *File System Implementation* section.
- dentry - an object with a string name, a pointer to an inode, and a pointer to the parent dentry. The picture below shows two dentries pointing to the same inode. In this case the files HOWDY and DOODY are said to be hard linked.




A symbolic link is a special file type whose data part carries a path to another file. The OS recognizes the data as a path, and redirects opens, reads, and writes so that, instead of accessing the data within the symbolic link file, they access the data in the file named by the link file. The picture below shows this - the symbolic link HOWDY opens the file DOODY via redirection when it is accessed.



## Performance

- See slides 33, 34, 40, 41 of chapter "File System Implementation"

## Disk scheduling

The goal of the operating system with respect to disk access to minimize service time by using hardware most efficiently.

- Time factors in reading (or writing) a disk sector are:
    1. seek time - the time for the disk are to move the heads to the cylinder containing the desired sector.
    2. rotational latency - the additional time waiting for the disk to rotate the desired sector to the disk head.
    3. transfer time - time to read data and move it to the system

- Schedule reads and writes to minimize seek time
    - do this by minimizing seek distance
    - a reasonably busy system or disk can have a significant impact on time
    - scheduling can be done by the system, the disk, or both

- Algorithms
    - First-come-first-served (FCFS) - serve disk read/write requests in the order in which they arrive. This is simple, fair, with acceptable performance if utilization is low. But it may result in excessive arm movement.
    - Shortest-seek-time-first (SSTF) - the request chosen to be serviced next is the one closest to the current head position. This could result in starvation if new requests are for head positions close to the current position. An SSTF schedule can be a considerable improvement over FCFS, but it is not optimal due to the possibility of starvation.
    - SCAN - The disk arm starts at one end of the disk, and moves toward the other end, servicing all possible requests along the way, then moves back toward the other end of the disk, doing the same. A SCAN scheduler can result in uneven service time - this is analogous to a windshield wiper - more raindrops are present near the ends of travel than in the middle of the windshield. No starvation for this scheduler.
    - C-SCAN - scans and services only while the arm is going in one direction. When the arm reaches the end it is moved to the other side of the disk without servicing anything and then resumes a service sweep. A more uniform service time is provided by this scheduler.
    - LOOK - like SCAN but the arm reverses direction when it services the last request while moving in the current direction.
    - C-LOOK - like C-SCAN but the arm reverses direction when it services the last request while moving in the current direction.
    - Performance depends on the number and types of requests.
    - Requests for disk service can be influenced by the file-allocation method.
    - The disk-scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm if necessary.
    - Either a modified SSTF (to prevent starvation) or LOOK is a reasonable choice for the default algorithm

## RAID

Use of multiple disk drives provides reliability via redundancy and may also improve performance. Several improvements in disk-use techniques involve the use of multiple disks working cooperatively

- Disk striping - uses multiple disks as one drive. Each data word is split into pieces, possibly as small as a bit, and the pieces go on different drives. If the failure rate for a disk is $r$ and the failure rate for a stiped set of $n$ disks is $1 - (1 - r)^n \approx n * r$ since one disk failure means total failure.
- Mirroring - keep a duplicate copy of a disk or stiped set. If $r$ is the failure rate for a disk or stiped set, then $r^2$ is the failure rate for the set plus a mirrored set. If there are $n$ mirrors then the failure rate is $r^{(n+1)}$.
- RAID levels
  - Level 0 non-redundant striping. Data is spread across all disks. There is no redundancy.
  - Level 1 mirroring. All disks are duplicated.
  - Level 2 memory-style error correction (Hamming) organization. Store two or more extra bits to detect and correct errors.
  - Level 3 bit interleaved parity organization. All parity bits are on one disk. There is no striping of data disks.
  - Level 4 block interleaved parity organization. All parity bits are on one disk. Data is spread across $n$ disks.
  - Level 5 block interleaved distributed parity. Parity bits and data are spread across all disks.
  - Level 6 $P + Q$ redundancy where $P, Q \in \{0, 1\}$. Stores extra parity to protect against multiple failures.