**Introduction**

An Operating System has two roles:

1. The OS is an **abstract machine**: hides hardware details, provide abstraction to programmers and applications.
2. The OS is a **resource manager**: ensure no starvation, progress for all processes, allocation according to desired policy.

The OS operates the same way as ordinary software does.

* It is a program that is executed + more privileges.
* Relinquishes control of the processor to execute programs, re-establishing control after system calls / interrupts.

The **Kernel** portion of an OS runs in **privileged mode** and mediates access to resources (e.g. CPU, RAM, IO Devices).

* Manages hardware on behalf of applications / provides abstraction.

The **User** portion of an OS interacts with the Kernel via. **System Calls**.

* OS gets control of an ap to see what they have requested, so then it can perform the requested operation.

A **Process** is an instance of a program running on a computer / entity that can be assigned and executed on a processor.

Contains three segments:

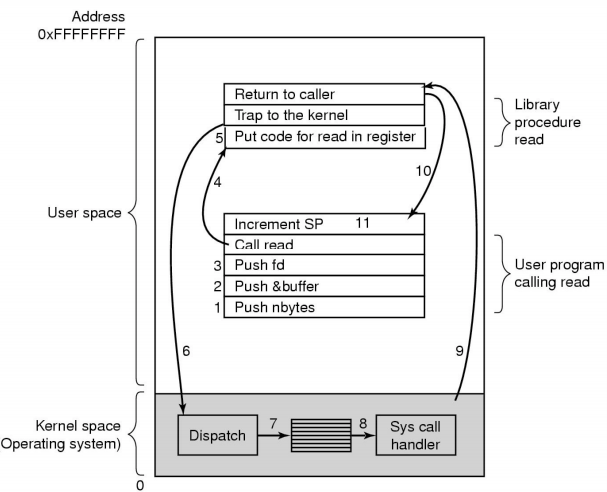
1. Text: executable program code / instructions.
2. Data: associated data i.e. global vars needed by the program.
3. Stack: execution context of the program i.e. local vars, program counters, stack ptrs, registers + more.

**Virtual Memory** provides users with a logical view of what is happening with physical allocation of memory in the machine at the time.

**File Systems** is basically a repository for data, were information is stored in named objects / files.

**System Calls**

**System Calls** is a mechanism to cross between the USER LEVEL and KERNEL LEVEL / OS. Logical workflow:

1. Application calls into system library, invoking a system function leading to controlled transition into the kernel.
2. While in kernel, perform a privileged operation.
3. Return to original caller the result.

**11 steps in making a System Call: read(fd, buffer, nbytes)**

User Program calling read()

|  |  |
| --- | --- |
| #1 #2 #3 | Push parameters on the stack. |
| #4 | Invoke the system call read(). |

Library procedure read()

|  |  |
| --- | --- |
| #5 #6 | Put code for read() on the register so the OS knows what to look at + **Trap** (switch) to the kernel. |
| #7 | **Dispatcher/Scheduler** receives control in kernel mode as a result of the trap, then performs a context switch.  **Context Switching**: Dispatcher saves state of previous process that was running then loads the initial/previously saved state of the new process. |

Kernel space actions.

|  |  |
| --- | --- |
| #8 | **Handler** saves register values onto the Kernel Stack + performs validations.  It looks at the **System Call Table** (map of syscall numbers and appropriate syscall) to invoke the correct system call function.  After performing the requested actions, the handler restores register values from the Kernel Stack and places the System Call return value on the User Stack. |
| #9 | Process goes back into User Mode at the saved return location. If return value indicated an error, then **errno** (global var) is set that indicates the status of execution. |

**Function Stack Frames + Exceptions**

Each function call allocates a new **Stack Frame** for local variables, the return address, previous frame ptr etc.

**REMEMBER: STACK GROWS FROM BOTTOM -> UPWARDS**

* **Frame Pointer** is the start/base of the current function’s stack frame.
* **Stack Pointer** is the end of the current function’s stack frame.

The compiler:

* Manages the stack of the particular thread of execution that the code is generated for.
* Knows how many local vars are being used and if extra space is needed on the stack or not.
* Will keep track of a Stack Pointer in one of the registers.

**Process control registers** are located in CP0;

* **c0\_cause** **register**: tells you what causes the exception.
  + ExcCode section: contains a value that gives the OS an indicator of what it was that triggered the exception.

e.g. ExcCode(8) = syscall.

* **c0\_status register**: tells you what the status of the CPU is / was.
* **c0\_epc register**: contains the address of the instruction of where to return to.

**Exceptions** are a sudden change in control flow in response to a change in processor state.

There are four classes of exceptions:

1. Interrupts: caused by signal from I/O device.
2. Traps: caused by applications requesting an OS service (call of read() etc.)
3. Faults: Application program causes a possibly recoverable error.
4. Aborts: Non-recoverable error.

Execution walkthrough with an exception – at the machine/hardware level:

1. Interrupt occurs as the previous instruction completed – right before start of a new instruction.
2. Currently in USER MODE, with INTERRUPTS ON.
   1. c0\_epc stores the **Exception Program Counter** so that the OS can find it
   2. c0\_status bits |KUc|IEc| are set to 1.

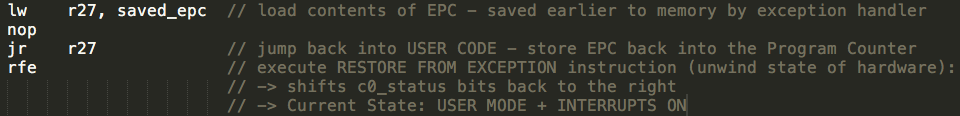
*^current state: USER MODE + INTERRUPTS ON*

* 1. c0\_status bits |KUc|IEc| are shifted two bits to the left -> |KUp|IEp| are now set to 1 / |KUc|IEc| are 0.  
     *^previous state: USER MODE + INTERRUPTS ON -> ^current state: KERNEL MODE + INTERRUPTS OFF*

1. c0\_cause is set with the ExcCode. Since it is an interrupt exception, ExcCode = 0.
2. Program Counter is set to 0x8000080 (2gigs + 80 bytes), the address of the general exception vector placed in the program counter.
   1. The application is now back in control and knows:  
      Where the app was previously | State of the system prior to the exception | What triggered the exception.

Returning from an exception:

* 1. Prior to exiting out of Kernel mode, the last the the OS will do is run:



**Processes**

**Processes**: Execution of an individual program.

**Dispatcher/Scheduler**: Gives control of CPU to the process selected by the **short-term scheduler**. Functions include:

* Context Switches: Dispatcher saves the state of prev process/thread + loads initial/prev saved state of the new process.
* Switching to User Mode
* Jumping to the proper location in the user program to resume the program indicated by its new state.

|  |  |
| --- | --- |
| Process Creation | Process Termination |
| * System initialisation: Foreground + Background processes * Execution of process creation syscall: i.e. new login shell for incoming SSH connection. * User request to create a new process. * Initiation of a batch job. | * Normal Exit: Voluntary * Error Exit: Voluntary * Fatal Error: Involuntary * Killed by another process: Involuntary |

A **Process Control Block (PCB) / Context Block** stores all info needed to keep track of a single process:

* A data structure in the OS Kernel for every process, identified by a *int process\_ID (PID)*

A **Process Table** is formed from all PCB’s:

* Facilitates context switching, scheduling and other activities.

**Threads**

**Thread**: A unit of execution that belongs to a program.

Thread Models: 3 ways to construct a program

1. **Threads Model**: Parallelism, blocking system calls.
2. **Single-Threaded Model**: No parallelism, blocking system calls.
3. **Finite-State Machine Model**: Parallelism, non-blocking system calls, interrupts.

Why Threads?

* Simpler to program than a state machine.
* Less resources are associated with them than a complete process.
  + Cheaper to create and destroy.
  + Shares resources (especially memory) between them.
* Performance: Threads waiting for I/O can be overlapped with computing threads.
  + If all threads are **Compute Bound**, then there are no performance improvements. i.e. SINGLE CPU.
* Take advantage of parallelism on machines with more than one CPU i.e. QUAD CORE CPU.

**Concurrency and Synchronisation Primitives**

**Race Condition**: Occurs when two or more threads access shared data and try to change it at the same time.

* Thread scheduling can swap between threads can any time. You don’t know the order in which threads attempt to access the shared data.
* Result of change in data is dependent on the thread scheduling algorithm.

**Critical Region:** Region of code where shared resources are accessed.

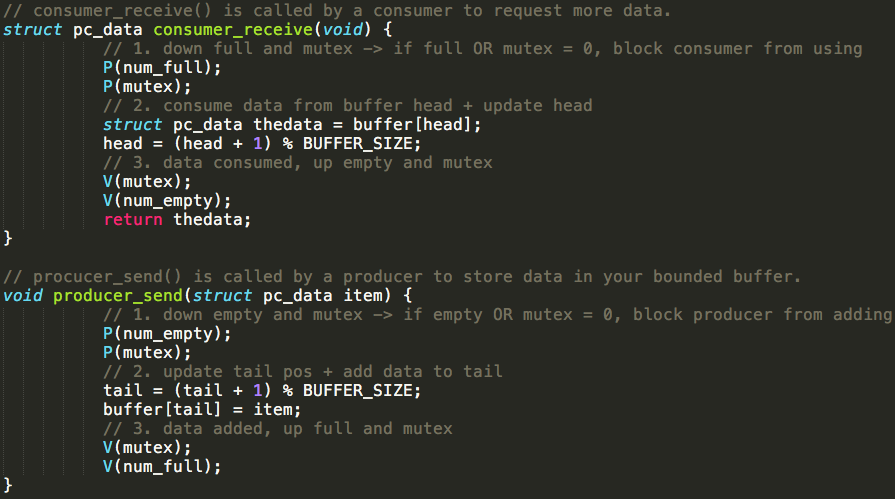
Conditions required of any solution to the critical region problem:

1. Mutual Exclusion: no two processes are simultaneously in a Critical Region
2. No assumption made about speeds / number of CPU’s.
3. Progress: No process running outside the Critical Region may block another interested process from entering when the region is free.
4. Bounded: No process waits forever to enter its Critical Region.

**Bounded Waiting**: Limit on number of times processes can enter a CR, then the system must grant access to other waiting processes. Prevents **Starvation**.

**Producer-Consumer Problem (Bounded Buffer Problem)** describes two processes, the producer and consumer who share a common, fixed-size buffer used as a queue. The producer’s job is to generate data (put it in the buffer) and at the same time, the consumer is consuming the data (remove it from the buffer) one piece at a time.

* Problem: Make sure the producer won’t add to a full buffer + consumer wont take away from an empty buffer.
* Solution: 2 sempahores + 1 mutex



**Semaphore** is a synchronisation primitive used to control access to a shared resource by multiple processes.

* If the resource is not available, block process and put it into a Process Queue until it is signalled to resume.

Semaphore operations:

**P(Semaphore S)**: [wait] semaphore value--. If value < 0, block process and put in process queue.

**V(Semaphore S)**: [signal] semaphore value++. Transfers blocked processes from the process queue to the ready queue.

**Mutex** (Mutual Exclusion Object) is similar to a semaphore except only the thread that holds the Mutex lock may unlock it.

* Semaphore implementation of a Mutex = initialise Semaphore value = 1.

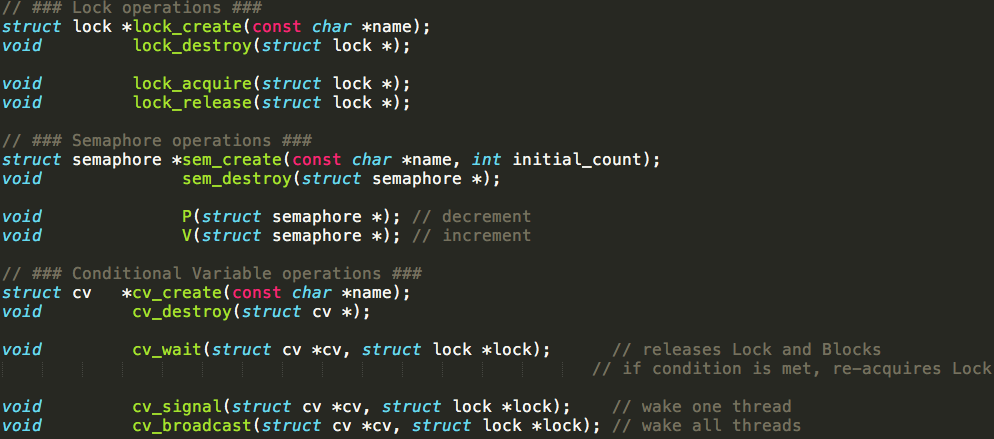
**Monitors** provide threads with a mechanism for temporarily giving up exclusive access to a shared resource in order to wait until a condition is met, before gaining exclusive access and resuming their task.

* Consists of a Mutex and a Conditional Variable.

**Condition Variables** is a queue of threads associated with a monitor.

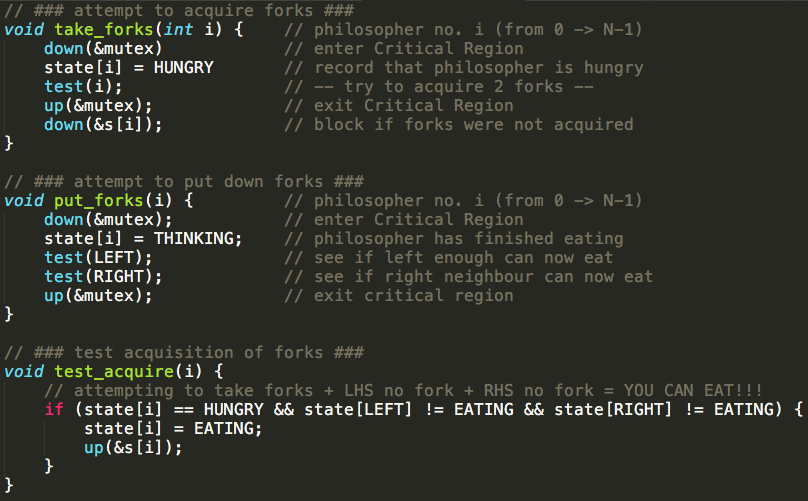
* While a thread is waiting on a condition variable, it does not occupy the monitor so other threads may enter the monitor to change its state. Has both **signal** (wake one thread) and **broadcast** (wake all threads) operations.

Summary implementation of synch primitives:



**Dining Philosopher’s Problem**: K philosophers seated in a circle with one chopstick between each pair of philosophers. A philosopher may eat if he can pick up two chopsticks next to him. Each chopstick may only be used by one person at a time. Infinite supply and demand of food is assumed. SOLUTION:

* Semaphore Array: one semaphore for each chopstick to control the behaviour of each philosopher.
* Mutex: enforce mutual exclusion so no two philosophers can access pick up / put down CR at the same time.



**Deadlocks**

**Deadlocks**: A set of processes is deadlocked if each process in the set is waiting for an event (usually the release of a held resource) that only another process in the set can cause.

Four conditions for deadlock:

1. **Mutual Exclusion**: several processes cannot simultaneously share a single resource.  
   (shareable resources like read-only file do not require mutually exclusive access thus cannot be involved in deadlocks)
2. **Hold and Wait Condition**: hold the resource I have + wait for the resource that I don’t have yet.
3. **No** **Pre-Emption Condition**: resources cannot be taken away pre-emptively by another process / OS.
4. **Circular Wait Condition**: must be a circular chain of 2 processes or more.

Strategies for dealing with Deadlock:

**1. The Ostrich Algorithm**: strategy of ignoring potential problems on the basis that they are exceedingly rare.

* Strategy is reasonable if deadlocks occur rarely + cost of prevention is high.
* Trade off between Convenience (Engineering) vs. Correctness (Mathematical).

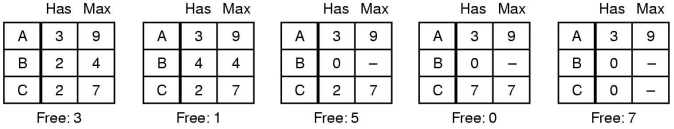
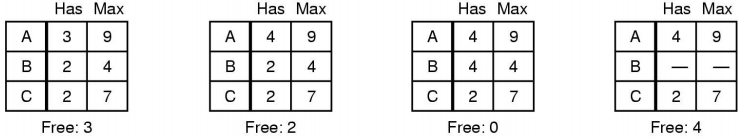
**2. Deadlock Prevention**: prevent one of the four conditions for deadlock

* Mutual Exclusion: not feasible to prevent – some devices/resources are not shareable such as a printer.
* Hold-and-Wait: must guarantee that whenever a process requests a resource, it does not hold other resource
  + STRATEGY: Processes must request and be allocated all resources before it begins execution.
  + Prone to **Livelock**: set of processes that are not blocked, run regularly, but never make progress.
* No Pre-emption: Not feasible to prevent. Can’t take away resources from a printer halfway through printing.
* Circular Wait Condition: **Resource Ordering** is a common technique in practice.

**3. Deadlock Detection and Recovery**: let deadlocks happen and come up with a recovery mechanism.

* Resource types w/ single instance: look for cycles within the graph, denoting a deadlock.
* Resource types w/ multiple instances: use a detection algorithm
  + STEP #1: Mark rows/processes which are EQUAL or GREATER THAN what is available
  + STEP #2: Update resources + Repeat process above
  + STEP #3: If all rows/processes can be marked off, the system is NOT deadlocked, else it is deadlocked.
* Deadlock recovery: recovery through killing one of the processes in the deadlock cycle.

**4. Deadlock Avoidance**: Avoid deadlocks, by allocating resources only if the resulting state = SAFE STATE

* Safe State: A system can allocate all resources to processes (up to their max) without entering into deadlock.  
  
* Unsafe State: Are not necessarily deadlocked, but we cannot guarantee that they will not deadlock/complete.  
  
* **Resource Algorithm: Current Allocated (C) + Resources Available (A) = Resources in Existence (E)**
* **Banker’s Algorithm**: Keeping the bank in a Safe State where all customers can request to borrow up to their credit limit at the same time.
  + Customers wishing to borrow such that the bank would enter an UNSAFE STATE must wait until somebody repays their loan, enabling transactions to be safe.
  + Allows: mutual exclusion, wait and hold, no pre-emption | Prevents: circular wait condition
  + Algorithm:

1. Calculate **Available (A) = Existence (E) – Current Allocated (C)**
2. Find a row/process in **Requested (R)** matrix where **R <= A**
3. IF row/process exists, it may be completed with the available resources.

ELSE an eventual deadlock is possible.

1. Pretend the selected process has acquired its requested resources, executed, terminated and **returned resources back to Available (A)**.
2. Repeat steps 2 and 3 until all processes have successfully reached termination (implies initial state was safe)  
   OR deadlock is reached (implies initial state was unsafe).

* Bankers Algorithm is not commonly used in practise as it is difficult to know in advance:  
  *The resources a process will acquire* or the *number of processes in a dynamic system*.

**Starvation** occurs when a process cannot attain a resource because the scheduler always allocates resources to other processes, denying the starved process from progressing.

* E.g. in a system where scheduler prefers smaller jobs, large jobs could never be chosen to attain resources
* Solution: First-Come, First-Serve policy.

**Processes and Threads Implementation**

Implementing Threads at the User-Level

* **User-Level Thread Control Block (TCB)** + **Ready Queue** + **Blocked Queue** + **Dispatcher**
* Kernel / OS has no knowledge of the threads (it only sees a single process)
* Thread management (created, exit, yield, wait) is implemented in a runtime support library.

Implementing Threads at the Kernel-Level

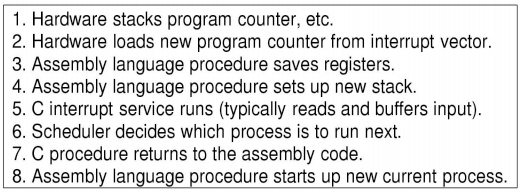
* **Kernel-Level Thread Control Block (TCB)** are stored in the kernel-level.
  + A subset of information (related to execution context) in a traditional PCB (Process Control Block).
  + TCB’s have a PCB associated with them of the process that the thread lives on.
* Thread management calls are implemented as system calls e.g. create() wait() exit()

|  |  |  |
| --- | --- | --- |
|  | Thread Implementation @ User-Level | Thread Implementation @ Kernel-Level |
| PROS | * Thread management + Switching much faster than at kernel-level. (No need to trap into kernel and back to user to switch) * Dispatcher algorithm can be tuned to the application.   (programmer can choose scheduling mechanism)   * Can be implemented on any OS. * Can support huge numbers of threads on a per-application basis. -> Use normal application virtual memory -> Kernel memory more constrained. Difficult to efficiently support differing number of threads for different applications. | * **Pre-emptive Multi-threading**: Execution of a thread can be interrupted by another thread / externally. * Parallelism -> Can overlap blocking I/O with computation. -> Can take advantage of a multi-processor |
| CONS | * Threads have to yield() manually (no timer-interrupt delivery to user-level) -> **Co-operative Multithreading**: a model where once a thread is given control it continues to run until it explicitly yields control or blocks. (doesn’t respond to external events). -> Hence a single poorly designed/implemented thread can monopolise the available CPU time. * Does not take advantage of multiple CPU’s. -> In reality, it is still a single threaded process as far as the kernel is concerned. * If a thread makes a blocking system call (or takes a page fault), the process and all internal threads block   -> Can’t overlap I/O with computation. | * Thread creation and destruction + blocking and unblocking threads requires kernel entry and exit. -> A lot more expensive than user-level implementation. |

**Processes and Threads: Context Switching**

**Context Switching**: refers to either a switch between THREADS or PROCESSES, saving and restoring their state.

Lowest level of what an OS does when an interrupt occurs – a context switch:



Any switch between threads/processes can happen when the OS is invoked on a System Call / Exception / Interrupt

* A thread switch can happen between any two instructions.

Context switching must be transparent for threads/processes:

* When a thread/process is dispatched again, it should NOT notice that something else was running in the meantime.

Example Context Switch scenario:

1. Running in user-mode, SP points to user-level stack. Exception/Syscall/Interrupt occurs and we switch to Kernel Stack.
2. We push a trap frame on the stack (saves state, including user-level PC and SP).
3. Call ‘C’ code to process Exception/Syscall/Interrupt: “C activation stack” builds up.
4. The kernel decides to perform a context switch.
   1. It chooses a target thread/process. It pushes remaining kernel context onto the stack.
5. Any other existing thread must be in Kernel-mode and have a similar stack layout to the one we are using.
6. We save the current SP in the PCB (or TCB) and load the SP of the target thread.
7. Load the target thread’s previous context 🡪 return to C / User-Mode

**Hardware Review: Memory Hierarchy, Caching and Performance**

**Registers** (FASTEST): holding results / variables.

**Cache**: holding memory which is being accessed again and again.

-> Cached for fast access.

-> L1 Cache: accessed without any delay

-> L2 Cache: takes more clock cycles than L1

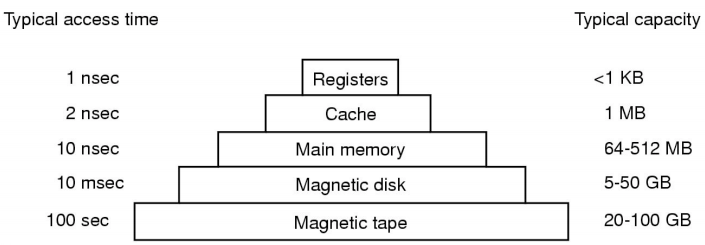
-> L3 Cache: takes more clock cycles than L2

**Main Memory or RAM**: hardware component

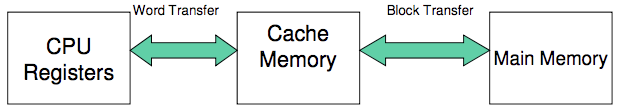
**Hard Disk**: data is kept permanently here. Not directly accessed by CPU

**Magnetic Tape**: used for backing up large data. Mount/unmounted

Computer Memory Hierarchy

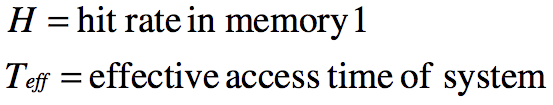
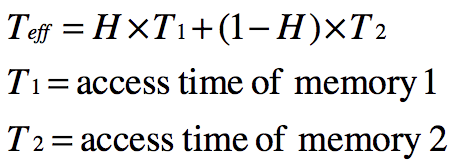


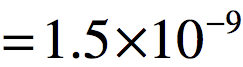
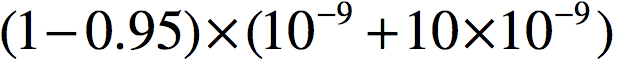
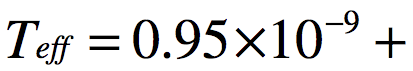
**CPU Cache** a hardware cache used by the CPU to reduce average cost to access data from the Main Memory.



* Holds recently used data or instructions to save memory accesses.  
  (cache cost = 1 to few cycles access time vs. Main Memory/RAM cost = 10’s – 100’s of cycles)
* Is hardware maintained and mostly transparent to software.
* Cache size ranges from a few kB to 10’s of MB.
* **Block Transfers (Cache Blocks)**: data is transferred between Main Memory and Cache in blocks of fixed size.
  + When a Cache Block is copied from memory into the cache, a **Cache Entry** is created, which includes copied data + requested memory address.
  + When the CPU needs to read/write to an address in Main Memory, it checks for the corresponding entry in the Cache with the memory address.
  + If the CPU finds the memory address in the cache, a Cache Hit has occurred. If not, a Cache Miss has occurred.
  + Cache Hit = CPU immediately reads/writes data to the Cache Block.
  + Cache Miss = cache allocates a new entry and copies data from the Main Memory, then the request is fulfilled from the contents of the cache.

Calculating **Effective Access Time**:



* Example: Cache memory access time = 1ns | Main memory access time = 10ns | Hit rate = 95%
  + 

**Moving Head-Disk Mechanism**: disk storage device which one or more read/write heads are attached to a moveable arm which allows each head to cover many tracks of information.

* Disks can read/write data relatively fast: *15,000rpm drive = 80MB/s* | *1KB block is read in 12 micro-seconds*.
* Access time is dominated by time to locate the head over data.
  + Rotational Latency: half rotation = 2 milliseconds
  + Seek Time: Full inside -> outside = 8 milliseconds
  + 2 milliseconds = 164KB in “lost” bandwidth.
* Strategy to improve system performance: AVOID WAITING FOR DISK ACCESS.
  + *CPU REGISTERS (FAST) ⇔ MAIN MEMORY/RAM (FAST) ⇔ HARD DISK (SLOW)*
  + Keep a subset of disk’s data in Main Memory (RAM): OS uses main memory as a cache of disk contents.
* Application approach to improving system performance: AVOID WAITING FOR INTERNET ACCESS.
  + *BROWSER (FAST) ⇔ HARD DISK (FAST) ⇔ INTERNET (SLOW)*
  + Keep a subset of internet’s data on disk: APPs use disk as a cache of the internet.

**File Management**

**File Structure Abstractions:**

* [LOWEST] Files as Byte Sequence 🡪 Files as Record Sequence 🡪 Files as Keys / Tree Structured [HIGHEST]
* Files as Bytes:
  + OS considers a file to be unstructured. Simplifies file management for the OS.
  + Applications can define their own structure. Use by UNIX/WINDOWS/modern OS’s.
* Files as Records:
  + A collection of bytes treated as a unit. E.g. Employee / Student record. A file is a collection of similar records.
  + Operations at the record level: read\_rec() write\_rec()
  + OS can optimise operations on records.
* Tree of Records:
  + Records of variable length, each with an associated key. Record retrieval is based on the key.
  + Used on some data processing systems (mainframes) / mostly incorporated into modern databases.

**File Types:**

* Regular files, Directories, Regular File types (ASCII, binary etc.)
* Device Files
  + Character Devices: un-buffered access to hardware devices, no control of read/write size. Defined by hardware
  + Block Devices: buffered access to hardware devices allowing programmers to read/write a block of any size.

**File Access Types**:

* Sequential Access: Read bytes/records from the beginning. Cannot jump around, but can rewind. (CASETTE – wind)
* Random Access: Read bytes/records in any order. Essential for database systems. (A RECORD PLAYER – move needle)

**File Attributes**: Protection (RWX), Password, Creator, Owner, Record Length, Creation Time, Last Accessed and more.

**Typical File Operations**: create(), delete(), open(), close(), read(), write(), append(), seek(), get\_attributes(), rename() and more.

**File Organisation and Access:** Programmer’s perspective

* Given an OS supporting unstructured files that are a stream of bytes, how can one organise the contents of the files?
* Possible access patterns: read whole file, individual blocks or records, retrieve a set of records, write a whole file sequentially, insert/update records in a file, update blocks in a file.
* Programmers are free to structure the file to suit the application.
* Criteria for file organisation: Rapid Access, Ease of Update, Economy of Storage (minimum redundancy of data)

**File Directories**: Provide mapping between file names and files themselves.

* Contain info about Attributes, Location, Ownership.
* Directory itself is a file owned by the operating system.

**Typical Directory Operations:** create(), delete(), opendir(), closedir(), readdir(), rename(), link(), unlink()

**File Sharing** **issues:** Access Rights + Management of Simultaneous Access

* Access Rights: **drwxrwxrwx** [ dir/reg file | user | group | owner ]
* Simultaneous Access: Most OS’s provide mechanisms for users to manage concurrent access to files  
  ( flock(), lockf(), system calls)
  + Typically users may lock entire file when it is to be updated.
  + User may lock individual records (i.e. ranges) during the update.
  + Mutual exclusion and deadlock are issues for shared access.

**File System Internals**

Application 🡪 OS

* Syscall Interface: create() open() read() write() . . .

FILE DESCRIPTOR / OPEN FILE TABLES

* Keep track of files opened by user-level processes.
* Implement details of File System calls

VIRTUAL FILE SYSTEM:

* Unified interface to multiple File Systems

FILE SYSTEM:

* Hides physical location of data on the disk.
* Exposes directory hierarchy, symbolic file names, protection etc.

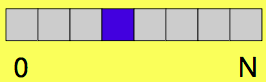
BUFFER CACHE / DISK SCHEDULAR: Provides optimisations.

* Cache keeps recently accessed disk blocks in memory.
* Scheduler schedules disk accesses from multiple processes for performance/fairness

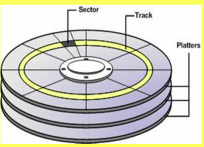
DEVICE DRIVER: Provides abstraction to hardware devices, enabling OS and programs to access hardware functions via. an interface.

* Driver communicates with devices through the *computer bus*.
* When a calling program invokes a routine in the driver, the driver issues commands to the device.

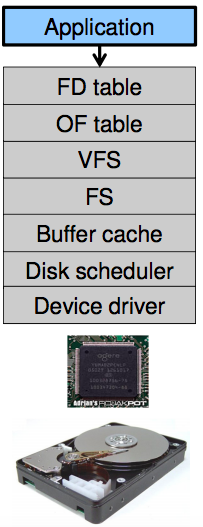
DISK CONTROLLER: Hides disk geometry + bad sectors. Exposes linear sequence of blocks



HARD DISK PLATTERS: Tracks + Sectors



**UNIX Storage Stack**:



Why are there many different File Systems?

* Different physical nature of storage devices i.e. optimised for CDROM / optimised for flask memory drives.
* Different storage capacities.
* Different CPU and memory requirements: modern PC’s vs. embedded devices.

Assumptions: We will focus on **Magnetic Disks** (the standard storage drives for consumer computers.

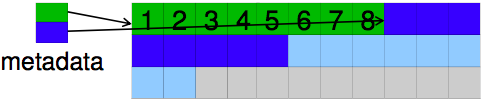
* Performance strategy: keep blocks that are likely to be accessed together close to each other.

Implementing a File System:

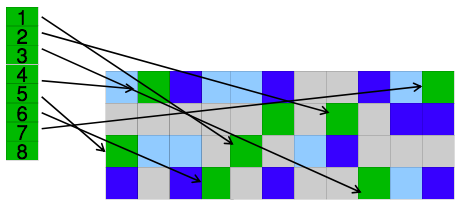
* FS must map symbolic file names into a collection of block addresses.
* FS must keep track of: which blocks belong to which files, what order the blocks are, which blocks are free for allocation
* Given a logical region of a file, the FS must track the corresponding blocks on disk.

File allocation methods:

* A file is divided into blocks (unit of transfer to storage)
* Given the logical blocks that make up a file, what method is used to choose where to put the blocks on disk?

**Contiguous Allocation**:

* Easy bookkeeping (need to keep track of starting block + length of file)
* Increases performance for sequential operations.
* Need the max size for the file at the time of creation.
* As files are deleted, free space becomes divided into many small chunks (*External Fragmentation*)

**Dynamic Allocation Strategies**:

* Disk space allocated in portions as needed. Allocation occurs in fixed-size blocks.
* NO external fragmentation.
* Does not require pre-allocating disk space.
* Partially filled blocks (*Internal Fragmentation*)
* Complex metadata management (maintain the list of blocks for each file)

**External and Internal Fragmentation**

* External Fragmentation: space wasted external to the allocated memory regions. Memory space exists to satisfy a request but it is un-usable as it is not contiguous.
* Internal Fragmentation: space wasted internal to the allocated memory regions (within blocks itself). Allocated memory may be slightly larger than requested memory.

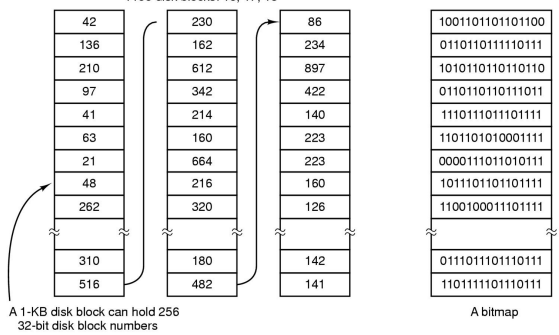
**Dynamic Allocation**: **Linked List Allocation**

* Each block contains a pointer to the next block in the chain. FREE BLOCKS are also linked in a chain.
* Only single metadata entry per file. Best for sequential files.
* Poor for random access, blocks end up scattered across the disk due to FREE BLOCK list eventually being randomised.

**Dynamic Allocation: File Allocation Table**

* Keep a map of the entire FS in a separate table.
  + A table entry contains the number of the next block of the file.
  + The last block of a file and empty blocks are marked using reserved values.
* The table is stored on the disk and is replicated in memory.
* Random access is fast (following the in-memory list)
* Issues: requires a lot of memory for large disks, free block lookup is slow.

**Dynamic Allocation: i-node based FS structure (i-node = index node)**

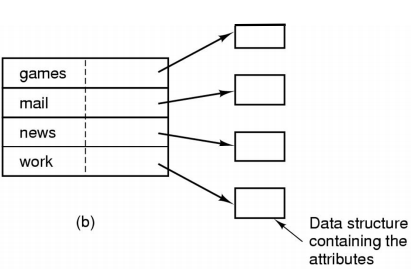
* Separate table/i-node for each file.
  + i-node = data structure that describes a file-system object such as a file / directory.
  + Each i-node stores the attributes and disk block location of the object’s data.
* Only keep table for open files in memory. Fast random access. The most popular FS structure today.
* Inode implementation issue: are allocated dynamically, hence free-space management is required for inodes.
  + Use fixed-size inodes to simplify dynamic allocation.
* Solutions to free-space management:
  + Approach 1: linked list of free blocks
  + Approach 2: keep bitmaps of free blocks and free inodes.
* **Free Block List**: a list of all unallocated blocks.
  + Background jobs can re-order list for better contiguity.
  + Store in free blocks themselves.
  + Only one block of pointers need to be kept in the main memory
* **Bit Tables** are individual bits in a bit vector/map, which flag used/free blocks.
  + 16GB disk with 512-blocks 🡪 4MB table.
  + May be too large to hold in main memory.
  + Expensive to search
  + Concentrating de-allocations in a portion of the bitmap has desired effect of concentrating access.
  + Simple to find contiguous free space.

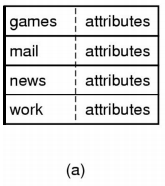
Implementing Directories: directories are stored like normal files.

* Directory entries are contained inside data blocks.
* A directory file is a list of directory entries. A directory entry contains file name, attributes and the file i-node number.
  + Maps human-orientated file name to a system-orientated name.

**Fixed-Size vs. Variable-Size directory entries**

* Fixed-Size: either too small or waste too much space.
* Variable-Size: freeing variable length entries can create external fragmentation in directory bocks.

**Searching Directory Listings**: Locating a file in a directory

* Linear scan (implement a directory cache in software to speed-up search)
* Hash lookup
* Binary Tree (100’s of thousands of entries).

**Storing File Attributes**:

1. Disk addresses and attributes in directory entry.
2. Directory in which each entry just refers to an i-node

Trade-off in FS block size:

* Larger blocks require less FS metadata.
* Smaller blocks waste less disk space (less internal fragmentation)
* Sequential Access: larger block size, fewer I/O operations required.
* Random Access: larger block size, more unrelated data loaded. Spatial locality of access improves the situation.
* Choosing an appropriate block size is a compromise.

**The EXT2 (Second Extended) File System**

**I-Nodes**: each file is represented by i-nodes on disk

* Inodes contain fundamental file metadata: access rights, owner, block index table of a file.
* Each inode has a unique number (system designated)
* Directories map file names to inode numbers: map human-orientated to system-orientated names.

**Inode Contents**

MODE: DRWXRWXRWX

UID: User ID

GID: Group ID

ATIME: Time of last access | CTIME: Time created | MTIME: Time last modified

SIZE: Offset of the highest byte written.

BLOCK COUNT: Number of disk blocks used by the file. Files can be sparsely separated.

* Only need to store the start/end of file, not all the empty blocks in between.

DIRECT BLOCKS: Block numbers of first 12 blocks in the file.

* Most files are small, so we can mostly find blocks of a file directly from the inode.
* Example: File with 11 blocks split up among other blocks.

REFERENCE COUNT: Keeps track of number of links/ptrs

to the specified inode, so file system knows to delete

the inode (underlying file or dir) when count = 0

**How to store files with data at offsets > 12 blocks?**

INDIRECT BLOCKS: Block number of a block

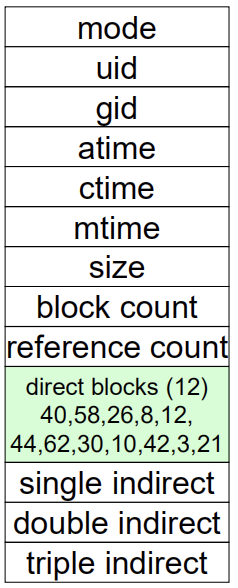
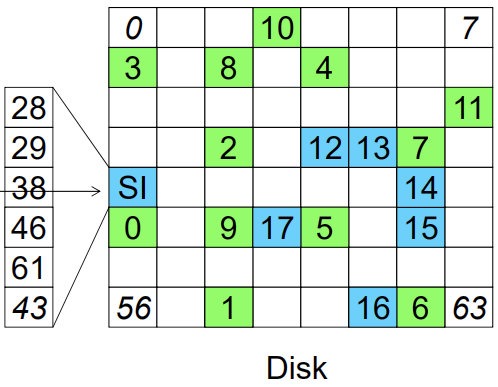
containing block numbers.

**Single Indirection**

* Requires 2 disk access to read:  
  1 for indirect block, 1 for the target block

**Double Indirection** + **Triple Indirection**

* As above + more inception



**Calculating Maximum File Size (with Single Indirection)**

Assuming 1Kb (1,000 bytes) block size and 4 byte block numbers, what is the max file size?

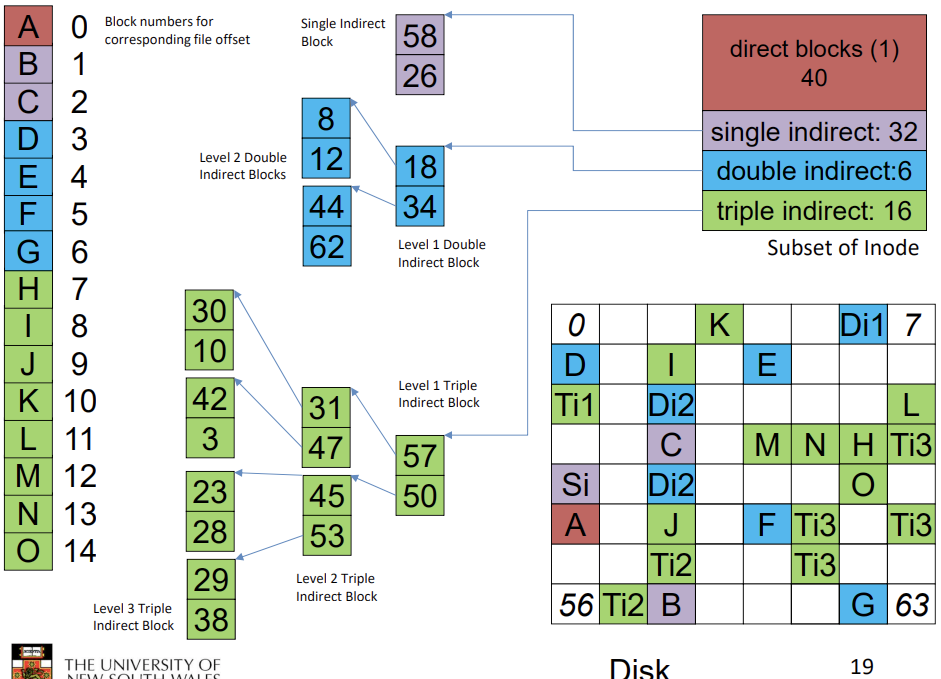
Direct Blocks = 12

Single Indirect Blocks = 1Kb/4 bytes = 256

Double Indirect Blocks = 256 \* 256 = 65,536

Triple Indirect Blocks = 256 \* 256 \*256 = 16,777,216

**MAX SIZE: 12 + 256 + 65,536 + 16,777,216 = 16,843,020 blocks \* 1Kb =~ 16GB**



**NOTE: 1024 bytes = 1 KiB (Kibibytes) | 1000 bytes = 1K (kilobytes / Kb)**

**Calculating Block Number for a specified byte address**

Assuming 4Kb blocks, 4 byte block numbers, 12 direct blocks.

EXAMPLE 1: Address = 1,048,576 bytes

**1. Calculate no. block numbers in indirect blocks:** 4Kb / 4b = 1024 block numbers

**2. Map out direct + indirect block ranges**:

|  |  |
| --- | --- |
| **Block Range** | **Location** |
| 0 - 11 | Direct Blocks |
| 12 - 1035 (11 + 1024) | Single Indirect Blocks |
| 1036 - 1,049,611 (1035 + 1024 \* 1024) | Double Indirect Blocks |
| 1,049,612 - ??? | Triple Indirect Blocks |

**3. Calculate Block Number:** 1,048,576 / 4,096 (block size) = Block #256

**3. Adjust for indirect offset**: #256 – 12 (single indirect offset) = **Block #244**

EXAMPLE 2: Address = 5,242,880 bytes

**Block Number:** 5,242,880 / 4,096 = Block #1280

**Double Indirect Offset:** #1280 – 1036 = #244

**Top 10 bits = 0 | Lower 10 bits = 244** => 0000000000 0011110100

**Inode Summary**

* The majority of files (small files within the 12 direct blocks) require only single access.
* Larger files require exponentially more disk accesses for random access: sequential access is still efficient.
* Can support really large files via. increasing levels of indirection.

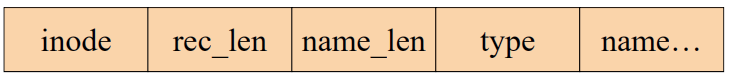
**System 5 Disk Layout – s5fs** (Where/How are Inodes stored?)

* 
* Boot Block: contains code to initialize the OS.
* Super Block: contains attributes of the File System itself
  + size, number of inodes, start block of inode array, start of data block area, free inode list, free data block list
* Inode Array: array containing inodes
* Data Blocks: all the blocks of data

**Issues with s5fs**:

* Long Seek Times: Inodes at the start of the disk, data blocks at the end. Must read inode before reading data blocks.
* Only one Super Block: Corrupt superblock = entire file system is lost.
* Block Allocation is Suboptimal: Allocation / de-allocation of blocks eventually randomises the free block list resulting in random allocation over time.
* Inode Access Suboptimal: inode free list also randomised over time, resulting in random inode access patterns.

EXT2 File System Directories



* Example #1: [ 7 | 12 | 2 | f, 1, 0, 0] = “f1” = inode 7
* Example #2: [ 43 | 16 | 5 | f, i, l, e | 2, 0, 0, 0 ] = inode 43

**Hard Links**: Hard links are pointers to an inode, so an inode can be represented by multiple names/pointers (hard links).

**Symbolic Links (software links)**: Symbolic links are independent files that contain a reference to another file or directory.

* Has its own inode + data block which contains a path to the target file.
* Marked by a special file attribute

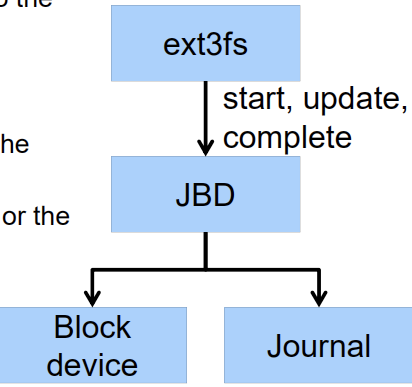
How does deleting a file work? (rm filename.txt)

* [ file1.txt ] [ file2.txt ] [ file3.txt ] => rm file2.txt
* [ file1.txt ] [ file3.txt ] => adjust the file1 record length to skip to the next valid entry (file3)

File System Reliability

* Disk writes are buffered in RAM. OS crash / power outage = lost data.
* Commit writes to disk periodically (e.g. every 30 seconds) + use the sync command to force a File System flush.
  + **File System Flush**: Transfer all temp data to permanent memory.
* File System operations are non-atomic (non-guaranteed return):
  + Incomplete transaction can leave the File System in an inconsistent state.
* **e2fsck**: scans the disk after an unclean shutdown and attempts to restore File System invariants.
* **Journaling File Systems**: Keep a journal of FS udpates -> before performing an atomic update sequence, write to journal -> replay the last journal entries upon an unclean shutdown.

**The EXT3 (Third Extended) File System**

Purpose of EXT3 is to add Journaling capability to EXT2 File System.

Provides backward / forward compatibility with EXT2: existing EXT2 can be mounted as EXT3.

Leverage the proven EXT2 performance.

Re-use most of EXT2 codebase + tools + e2fsck.

**The EXT3 Journaling Layer: Journaling Block Device (JBD) interface**

* JBD can keep the filesystem journal on a Block Device or in a File.
* Separation of Concerns: JBD is independent of EXT3-specific data structures.
* Code Re-use: JBD can be used by any other FS that requires journaling.

Transaction Lifecycle

IN PROGRESS / RUNNING

Updates are cached in RAM

COMPLETED / LOCKED

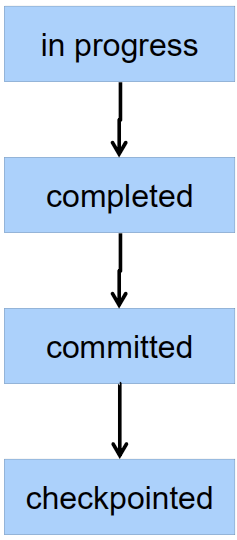
No additional updates are allowed in the same transaction.

COMMITTED: Write transaction data to the journal + mark as committed (persistent storage)

* Multiple File System transactions are committed in one go.
* Transaction can be replayed after an unclean unmount.

CHECKPOINTED: Flush the journal to the disk. Used when journal is full or FS is being unmounted.

* Updates are written to the file system.
* The transaction is removed from the journal.



**Virtual File Systems (VFS)**

Modern computers need to support multiple file system types: CDROM, MSDOS etc.

VFS provides a single system call interface for many file systems:

* File-based interface to arbitrary device drivers *( /dev )* + File-based interface to kernel data structures *( /proc )*
* Provides an indirection layer for system calls.

Two major data types:

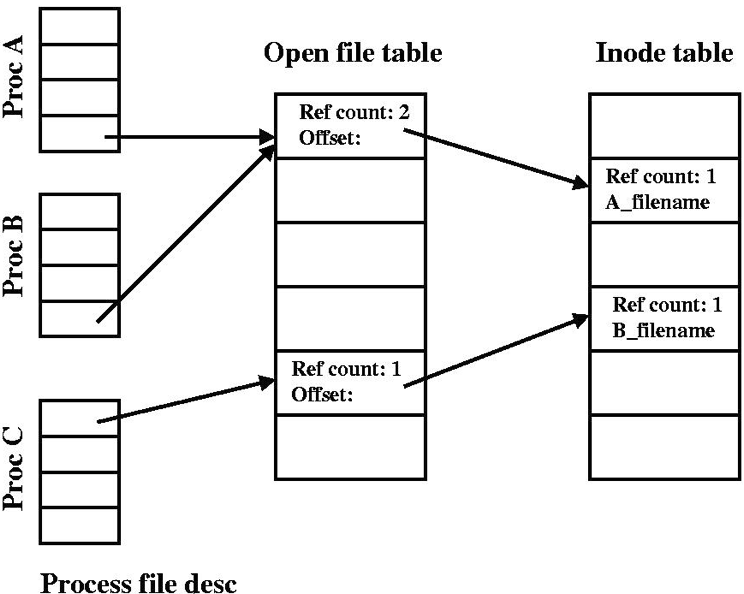
1. **VFS**: Represents all file system types.
   1. Contains pointers to functions to manipulate each file system as a whole (e.g mount/unmount).
2. **Vnode**: Represents a file (inode) in the underlying File System.
   1. Points to a real inode, contains ptrs to functions to manipulate files/inodes (e.g. open, close, read, write etc.)

**File Descriptors**: an entry (non-negative int) created by the OS used to access a file or other input/output resource.

* Each open file as a File Descriptor. Read/Write/Seek use them to specify which file to operate on.
* State associated with a file descriptor:
  + File Pointer: determines where in the file the next read or write is performed.
  + Mode: Was the file opened read-only etc.

**Per-Process File Descriptor Table with Global Open File Table**

* Per-process file descriptor array: contains pointers to Open File Table Entry.
* Open File Table Array: contains entries with a file pointer + pointer to a vnode.
* Provides shared AND independent file pointers if required.



File Descriptor Table (per process): Array of pointers (identifiers of files opened by the current process) to the Open File Table.

* Whenever a process creates a file, it gets an index from this table.

Open File Table: Contains info that is global to the kernel.

* Each entry contains a file offset where the user’s next read/write will start + access rights + pointer to a vnode in the vnode Table.

When 2+ processes open a file, there’s an entry in the OFT per open() call. There can even be multiple entries in the OFT for one process opening a file multiple times.

A single entry is NOT created in the OFT for different processes opening the same file (just one entry in vnode table)

**Buffer**: temporary storage used when transferring data between two entities.

* Good when entities work at different rates.
* Good when unit of transfer is incompatible. E.g. between Application Programs and Disk

**Buffering Disk Blocks (Disk Buffer)**: allow applications to work with arbitrarily sized region of a file.

* Writes can return immediately after copying to kernel buffer. (avoids waiting until write to disk is completed)
* Can implement read-ahead by pre-loading next block on disk into kernel buffer. (avoids wait until next read is issued)

APPLICATION PROGRAM Transfer of arbitrarily sized regions of file 🡪

BUFFERS IN KERNEL RAM Transfer of whole blocks 🡪 DISK

**Cache**: fast storage used to temporarily hold data to speed up repeated access to the data.

File System Consistency

**UIO** **(User I/O)**: is a data structure representing a memory buffer to be used in a data transfer.

* Used in read/write interfaces to device drivers supporting char or raw I/O.
* Used in instances where input/output buffer exists in different address spaces.

**Struct IOVEC**: Useful for reading/writing data to multiple buffers, in a single kernel call rather than using multiple calls to read/write which is inefficient.

**Memory Management**

OS Memory Management:

* Keeps track of free/used memory and allocates/de-allocates free memory to processes
* Manages transfer of memory between RAM and disk.

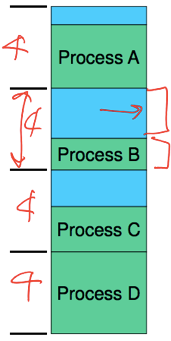
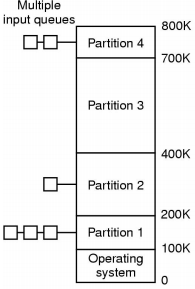
Programmers ideally want memory that is **Fast, Large, Non-volatile** which is not possible

* Memory management focus is usually on **RAM <-> DISK** (i.e. **Electronic Disk <-> Magnetic Disk**)

**Swapping/Paging**: transferring processes to and from external storage during execution.

**Mono-programming**: simple memory management without paging/swapping. Running only 1 program at a time.

* Used mostly in embedded devices. Usually memory available = memory required.
* Poor CPU utilisation (I/O waiting) and memory utilisation.

How do we divide memory between processes?

**FIXED PARTITION STRATEGIES**

**Fixed Approach #1:** divide memory into fixed equal-size parts.

* Any process <= fixed size partition can be loaded in.
* **Internal Fragmentation:** Unused space in partition is wasted.
* Processes smaller than main memory, but larger than a fixed size partition cannot run.

**Fixed Approach #2**: divide memory into a selection of different-sized parts. Sizes are based on workload.

* Each partition has a queue. Processes are placed in a queue for the smallest partition that it can fit in.
* Processes wait for when assigned partition is empty to start.
* Issues: some partitions may be idle, small jobs are available but only large partitions are free.

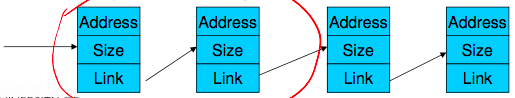
**Fixed Approach #3 (Alternative Queue Strategy)**: Single queue, search for any jobs that fit at all.

* Small jobs will fit into large partitions if necessary.
* Increases internal memory fragmentation.

**DYNAMIC PARTITION STRATEGIES / PLACEMENT ALGORITHMS**

Dynamic Allocation issues:

* **External Fragmentation:** Many small non-adjacent blocks of memory are formed, which can service a  
  memory request from a process if they are put together, but can’t be done since they aren’t together.
* **Relocation**: how does a process run in different locations in memory?
* **Protection**: how do we prevent processes from interfering with each other?



**Classic Approach**: Linked-list of available “holes” { mem\_addr, size, link->next }

* Are kept in order of increasing memory address to simplify merging of adjacent “holes” into larger ones.

**First-Fit Approach**: Scan the list from the beginning for the 1st entry that that is large enough.

* Intent: minimise amount of searching performed.
* Generally results in smaller holes at the front of memory that must be searched over when trying to find a free block.
* May have lots of unusable holes at the beginning = External Fragmentation.
* Tends to preserve larger blocks at the end of memory.

**Next Fit Approach**: Scan the list from the point in list where the last request succeeded.

* Spreads allocation more equally through entire memory, but often allocates a block at the end of memory where the largest block is found.
* The largest block is broken up into smaller blocks, unable to service larger requests + unable to fit in first blocks.

**Best Fit Approach**: Chooses the block that is closest in size to the request.

* Poor performer, as it has to search the complete list (does more work than First-Fit or Next-Fit.
* Since the smallest block fit is chosen for a process, results in least amount of External Fragmentation.  
  (Creates lots of unusable holes)

**Worst Fit Approach**: Chooses the block that is largest in size (worst fit).

* The idea is to leave a usable fragment left over.
* Poor performer, as it has to do more work to search the complete list. Does not result in significantly less fragmentation

First-Fit and Next-Fit are generally better than the other approaches and easiest to implement.

However, these have largely been replaced by more complex and specific allocation strategies.

**Compaction**: reducing External Fragmentation by shuffling memory content to place all free memory together in 1 large block.

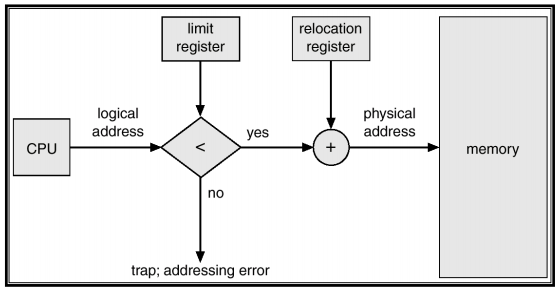
* This only works if we can relocate running programs + generally requires hardware support.

When can logical (virtual) addresses bind to physical memory addresses?

* **@ Compile/Link Time**: Logical->Physical binding when your code is being processed by the compiler / when a compiler is transforming your code into an executable binary.
  + If location changes, will need to recompile.
* **@ Load Time**: Logical->Physical binding when an OS is reading an executable from long-term storage (DISK) and loading it into short-term memory (RAM) from which it can be executed.
  + Hard-drive is too slow to feed the CPU, so fast memory (RAM) is used to store instructions/program.
* **@ Run (execution) Time**: Logical->Physical binding when a program is executing or running. Most OS’s use this method.
  + Additional memory may be allocated / de-allocated at this time.

Hardware Support for Runtime Binding + Memory Protection

* **Base Register:** Need an offset/base to its logical address = protect memory “lower” than the process.
* **Limit Register:** Need a maximum limit logical address = protect memory “higher” than the process.



* Base and Limit registers must be changed at **Load Time / Relocation (compaction) / Context Switch**
* Pros: Supports protected multi-processing / multi-tasking
* Cons: Physical memory allocation must still be contiguous, entire process must be in memory, does not support partial sharing of address spaces (no shared code, libraries or data structures between processes)

The above mechanisms are suited for a **Batch System**, where there are a limited number of dynamically allocated processes.

What about a **Timeshare System?** (sharing of resources by many processes at the same time).

* Timesharing need more than just a small number of processes running at once.
* In Timesharing, we need to support a mix of active and inactive processes, of varying longevity.

**Swapping**: process can be swapped temporarily out to a Backing Store and back into memory for continued execution.

* **Backing Store:** fast disk large enough to accommodate copies of all memory images for all users
* Can prioritize between lower-priority / higher-priority processes.
* **Transfer Time**: total transfer time is directly proportional to the amount of memory swapped.

**Overlay**: Keep in memory only instructions and data that are needed at any given time.

* A method that allows programs to be larger than the computer’s main memory.
* It is implemented by the user, with no special support needed from the OS.
* Embedded systems typically use an overlay method because of limitation to physical memory + lack of virtual memory.

**Virtual Memory**: Developed to address issues with the simple schemes.

* Two classic variants: **Paging** (dominant method) + **Segmentation**

**Virtual Memory Paging**:

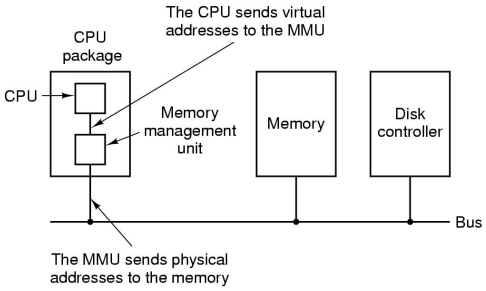
* PHYSICAL MEMORY is divided into small equal sized chunks called **Frames**.
* PROCESS’S VIRTUAL ADDRESS SAPCE is divided into same size chunks called **Pages**
  + Virtual memory addresses consist of a Page Number + Offset within the page.
* OS maintains a page table (virtual), containing the frame location (physical) for each page.
  + Used to translate each virtual address to physical address.
* A process’s physical memory does NOT have to be adjacent to each other.

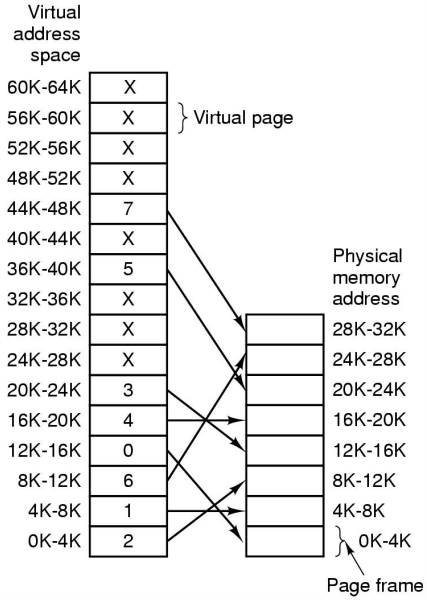
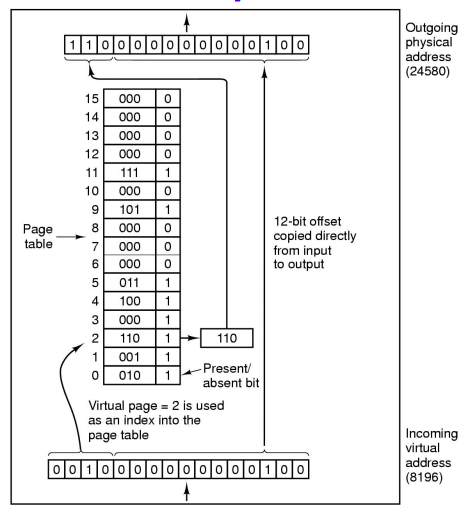
Benefits of Paging:

* No External Fragmentation. Small Internal Fragmentation (in the last page)
* Allows sharing by mapping several Pages to the same Frame.
* Abstracts physical organisation: programming only has to deal with virtual addresses.

**Memory Management Unit (MMU)**: also called **Translation Look-aside Buffer (TLB)**

Position + Function of the MMU:

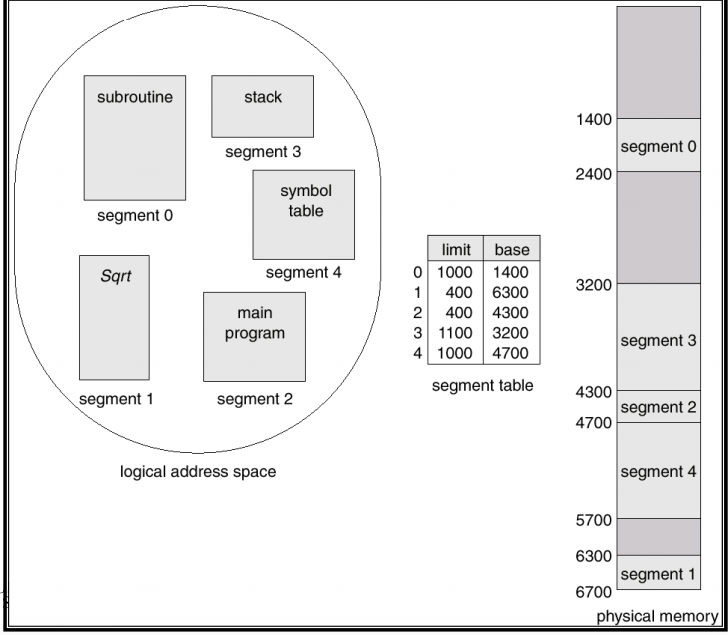


MMU Operations: Virtual Memory Paging Example:  


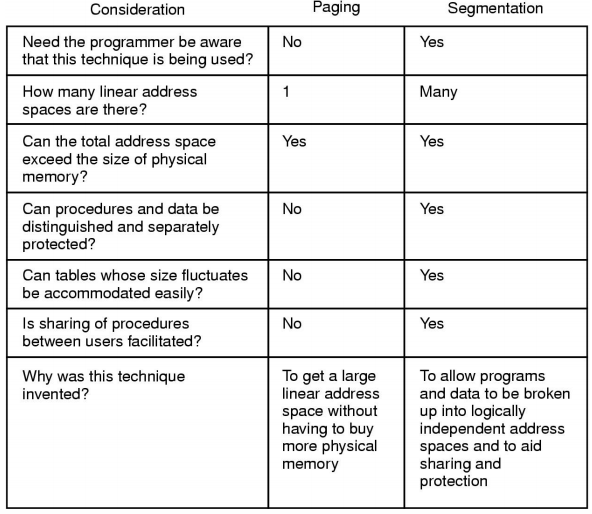
**Virtual Memory Segmentation**: Memory is allocated in various sizes (segments) depending on the need for address space by the user/process.

* Logical Address: consists of tuple **<segment-number, offset>**
* **Segment Table**: each entry has { base\_addr, limit }
* **Segment Table Base Register (STBR)**: points to segment’s starting physical location in memory.
* **Segment Table Length Register (STLR)**: indicates number of segments used by the program.
  + Segment # is illegal if S < STLR

Segmentation Example:



Comparison between Virtual Memory PAGING and SEGMENTATION:



**Virtual Memory (page-based)**

Shared Pages:

* Private code and data: each process has own copy of code and data + can appear anywhere in address space.
* Shared code: single copy of code shared between all processes executing it. Code must not be self-modifying.
  + Code must appear at the same address in all processes.

Page table structure: page table is an array of frame numbers, indexed by a page number.

* Each **Page-Table Entry (PTE)** has the following attributes:

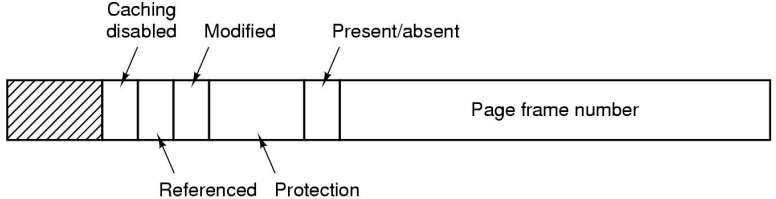
Present/Absent: indicates a valid mapping for the page.

Modified Bit: indicates if page has been modified in memory.

Reference Bit: indicates if the page has been accessed.

Protection Bit: RWX permissions

Caching Bit: indicates if processor should bypass the cache when accessing memory.



Virtual Memory Summary:

* Programs use **Virtual Addresses**
* Virtual -> Physical address mapping is done by the **Memory Management Unit (MMU)**
  + Check if page has present/absent bit set.
  + If YES: address in page table form most-significant bits in physical address.
  + If NO: request a non-loaded page / bring in page from disk (**Page Fault**)

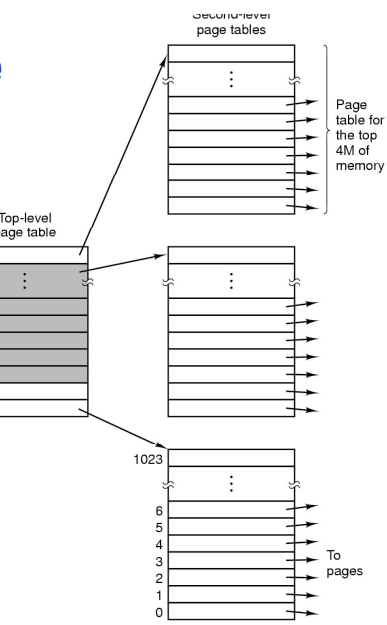
**GOOD EXPLANATION OF PAGE TABLES / INVERTED: https://www.quora.com/What-is-the-differences-between-a-page-table-and-an-inverted-page-table**

**Calculating size of Page Table (EXAM STUFF)**

Assume 32bit virtual address (address space: 2^32 = values), with 4kB page size.

* 4kB = 4,000bytes = **~2^12 bits / page (size)**
* 2^32 / 2^12 = **2^20 number of pages / frames possible (page size = frame size)**
* 20 bits required for frame number + 1bit (present/absent) + 1bit (modified) + 1bit (reference) + 1bit (prot) + 1bit (caching) = 25 bits = **4 bytes for each entry.**
* Page table size = 4 bytes \* 2^20 pages = 2^2 \* 2^20 = 2^22 bytes = 4,194,304 bytes = **~4MB Page Table size**

Page Table size issues:

* A page table is very large. Access has to be fast, as there is a lookup required for every memory reference.
* Where do we store the page table?
* **Page Tables are implemented as data structures in main memory**.
* Most processes do not use the full 4GB address space, so we need a compact representation that does not waste space but is still fast to search.

**Two-Level Page Table**: 2nd level page table represents unmapped pages and are not allocated.

* The top level page table entry shows NULL.

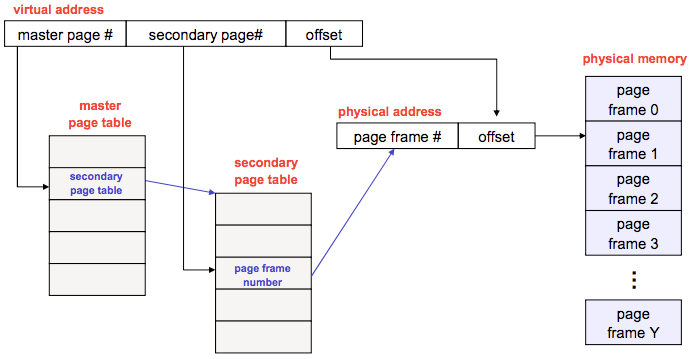
**Inverted Page Table**: entry for each PHYSICAL frame.

* 1 single Page Table for the entire system.

**RE-WRITE MORE CONDENSED NOTES SHOWING JUST THE  
ALGORITHMS FOR EACH PAGE TABLE, 2nd LEVEL PAGE TABLE,**

**INVERTED PAGE TABLE, HASHED PAGE TABLE**

**2 Level Page Table**



**Translation Look-aside Buffer**

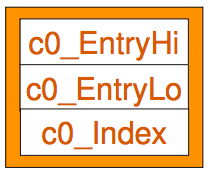
Reference Virtual\_Mem.c notes.

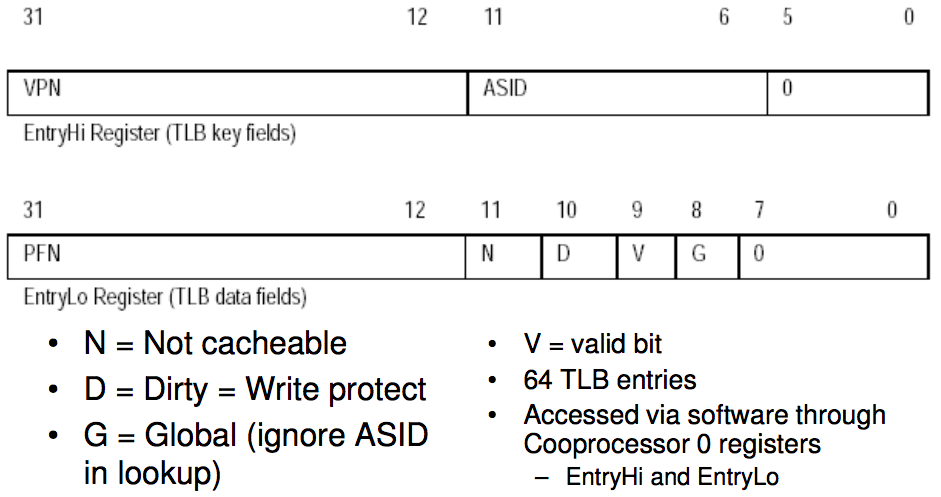
**MIPS R300 TLB**

TLB Entry structure: [ ENTRYHI | ENTRYLO ]

**c0 Registers (co-processor 0)**

Are used to read/write individual TLB entries.





**MIPS VM related Exceptions**

**TLB Refill Handling**

TLB refill is handled by software: an exception handler.

TLB refill exceptions accessing Kuseg are expected to be frequent

* CPU is optimised for handling refills by having a special exception handler just for TLB refills.

**Others (handled by a general exception header)**

TLB Mod: TLB modify exception, attempt to write to a read-only page.

TLB Load: Attempt to load from a page with an invalid translation.

TLB Store: Attempt to store to a page with an invalid translation.

**Demand Paging / Segmentation**

Only parts of a program image need to be resident in memory for execution.

Therefore, we can transfer unused pages/segments on to disk + reload these pages on-demand.

* Reload is triggered by a page / segmentation fault.
* Faulting process is blocked and another is scheduled.

**Principle of Locality / Locality of References**

Programs tend to use data and instructions they have used recently, so we can reasonably predict what instructions and data a program will use in the near future based on its accesses in the recent past.

* 90/10 Rule: a program spends 90% of its time in 10% of its code.

Two types of locality:

* Temporal Locality: recently accessed items are likely to be accessed in the near future.
* Spatial Locality: items whose addresses are near one another tend to be referenced close together in time.

**Working Set**

The Memory Working Set are the pages required by an application in a specified time window.

* If window too small => will not encompass the entire locality.
* If window too large => will encompass several localities.
* If window approaches infinity => will encompass the entire program.
* *Window Size is an application-specific trade-off*.

**Input / Output Management Intro**

**I/O Introduction**

Challenge with I/O management: having a uniform and efficient approach.

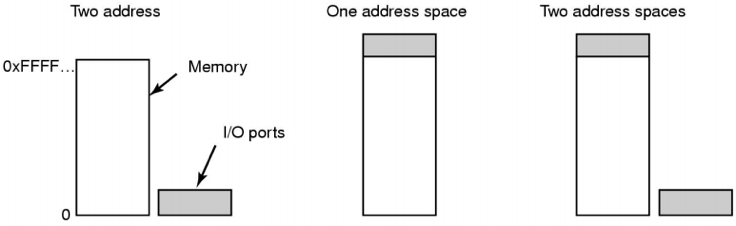
* Data Rates differ greatly between different types of I/O devices.

I/O device handling considerations:

* Complexity of Control
* Unit of Transfer: stream of bytes for a terminal / larger blocks for a disk.
* Data Representation: encoding schemes.
* Error Conditions: devices respond to errors differently / expected error rate also differs.
* Layering: devices that are the same but have different layers i.e. Hard-Disk, USB-disk, RAM-disk
  + Interaction of layers: swap partition and data on same disk
  + Priority: keyboard, disk, network

Three ways of accessing I/O Controllers:

1. Separately I/O and memory space: Controller registers appear as I/O ports, accessed with special I/O instructions
2. Memory-mapped I/O: Controller registers appear as memory. Use normal load/store instructions to access.
3. Hybrid: Both ports + memory mapped I/O.



Bus architectures:

* Single-bus architecture: Single bus where all addresses (memory and I/O) travel along.
* Dual-bus memory architecture: CPU reads/writes memory over a high-bandwidth bus, with a memory port to allow I/O devices access to memory.

I/O Controller Interrupts:

* Devices are connected to an Interrupt Controller via. lines on an I/O bus.
* Interrupt Controller signals interrupt to CPU and is eventually acknowledged.

**Types of I/O Interactions**

Programmed I/O (or Polling / Busy Waiting)

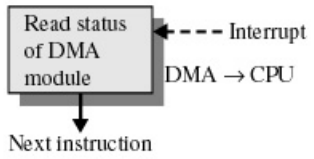
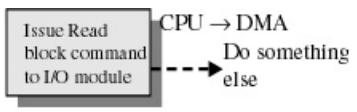
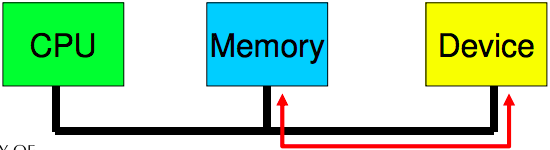
* I/O module (controller) performs the action, not the processor.
* Sets appropriate bits in the I/O status register.
* No interrupts occur.
* Processor checks status update until operation is complete: **wastes CPU cycles**

Interrupt-Driven I/O

* Processor is interrupted when I/O module (controller) is ready to exchange data.
* The processor is free to do other work, no needless waiting.
* Consumes a lot of processor time because every word read/written passes through the processor.

Direct Memory Access

* Transfers blocks of data directly between the Device and Memory, so the processor is not needed for copying.
* An interrupt is sent when the task is complete.
* The processor is only involved at the beginning and end of the transfer.



Direct Memory Access Considerations:

* PRO: Reduces number of interrupts = less expensive context witches or kernel entry-exits.
* CON: Requires contiguous regions (buffers).

Process to perform Direct Memory Access transfer of data:

1. Device driver is told to transfer disk data to a buffer at address X.
2. Device driver tells Disk Controller to transfer C-bytes from disk to buffer at address X.
3. Disk Controller initiates a DMA transfer.
4. Disk Controller sends each byte to the DMA controller.
5. DMA Controller transfers bytes to buffer at address X, increasing memory address and decreasing C until = 0.
6. When C = 0, DMA interrupts the CPU to signal transfer completion.

**CREATE CONDENSED NOTES BASED OFF THE LEARNING OUTCOMES + PRACTISE TUTE QUESTIONS AND FINALS**

**I/O Device Evolution: Complexity and Performance**

The general trend is to offload I/O processing to the I/O device itself.

* Reduced load on CPU and improved performance.

Example: I/O Processor

* I/O module has its own local memory, internal bus etc.
* It is a computer itself

**Input / Output Management**

**OS Design Issues**

Efficiency: Most I/O devices are slow compared to main memory and the CPU.

* Multi-programming allows for processes to be waiting on I/O while another process executes.
* I/O still cannot keep up with CPU speed.
* GOAL: optimise I/O efficiency, especially Disk & Networking I/O.

Generality/Uniformity: Ideally handle all I/O devices in the same way, both in the OS and user applications.

* Problem: Diversity of I/O devices, different access methods (random vs. stream based) as well as vastly different data rates. Generality often compromises efficiency.
* Hide most of the details of I/O in lower-level routines so that processes and upper levels see devices in general terms such as read() write() open() close() etc.

**I/O Software Layers**

Interrupt Handlers: can execute at almost any time

* Raises concurrency issues in the kernel.
* Can propagate to the userspace (signals) calling similar issues.
* Generally structured so I/O operations block until interrupts notify them of completion.  
  *^ look at kern/dev/lamebus/lhd.c* for an interrupt handler example.

Interrupt Handler Steps:

1. **Save Registers** not already saved by hardware interrupt mechanism.
2. (optional) **Set up context** for interrupt service procedure. Typically, the interrupt handler runs in the context of the currently running process – no expensive context switch is required.
3. **Set up stack** for interrupt service procedure. Handler usually runs on the kernel stack of current process.
4. **Ack/Mask interrupt controller**, re-enable other interrupts.
5. **Run interrupt service procedure**: acknowledges interrupt at device level. Figures out what caused the interrupt (e.g. received a network packet, disk read finished). If needed, it will signal the blocked device driver.
6. **In some cases, will have woken up a higher priority blocked thread**: choose newly woken thread to schedule next. Set up MMU context for process to run next.
7. **Load new/original process’ registers**
8. **Re-enable interrupt**: start running the new process.

Sleeping in interrupts: an interrupt generally has no context (runs on current kernel stack).

* It is unfair to sleep on interrupted process (deadlock is possible).
* Where do we get the context for long running operation?
* What goes into the ready queue?

We solve the above questions by handling interrupts by two halves: **Top and Bottom**

Top Half: routine that responds to the interrupt.

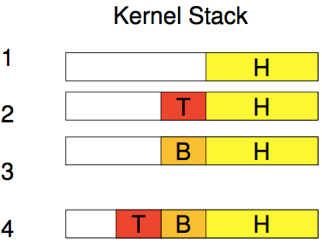
* Saves device data to a device-specified buffer, schedules the Bottom Half and exits.

Bottom Half: routine scheduled by the Top Half to be executed later, at a safer time (where all interrupts are disabled)

* Performs all other work that is required e.g. awakening processes, starting up another I/O operation etc.

This Top & Bottom Half setup enables the Top Half to service a new interrupt while the bottom half is still working.

Top & Bottom Half Stack Usage



1. Upper Software.

2. Interrupts processing. Interrupts are disabled.

3. Deferred processing. Interrupts are re-enabled.

4. Interrupt while in bottom half.

The interrupt handler defers work onto an in-kernel thread:

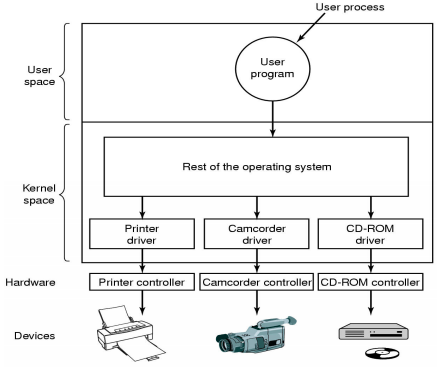
* In-Kernel thread handles deferred work: scheduled normally + can be blocked.
* Both low interrupt latency and blocking operations.

**Device Drivers**

Device Drivers in the past were compiled into the Kernel. The number / type of devices rarely changed.

Devices Drivers now are dynamically loaded when needed. Users (device installers) can’t build the Kernels themselves.

* The number and types of devices vary greatly. Works even while the OS is running e.g. hot-plug USB devices.



Drivers are classified into similar categories:

* Block devices and character (stream of data) devices.

OS defines a standard interface to the different classes of devices:

* Device specs often help e.g. USB.

Device Driver’s Job:

* A driver’s job is to translate requests through the device-independent standard interface open() close() read() write() into appropriate sequence of commands (register manipulations) for the particular hardware.
* Initialise the hardware at boot time and shut it down cleanly.

After issuing the command to the device, the device either:

* Completes immediately and the driver returns to the caller.
* OR device must process the request and the driver usually blocks, waiting for an I/O complete interrupt.

Drivers are **re-entrant** (thread-safe):

* They can be called by another process while a process is already blocked in the driver.
* **Re-entrant** = no global, dynamic data.

**Device-Independent I/O Software**

I/O software these days should be device-independent, where it should be possible to write programs that can access any time of I/O device without having to specify the type of device in advance.

Device independent software includes:

(1) Buffer or Buffer-cache management. (2) Managing access to dedicated services. (3) Error reporting.

**Driver <-> Kernel Interface**

Major issue is having a uniform interface to devices and the kernel.

* Uniform DEVICE interface for kernel code:
  + Allows different devices to be used the same way i.e. no need to rewrite file-system to switch between IDE or RAM disk.
  + Allows internal changes to device driver without fear of breaking Kernel Code.
* Uniform KERNEL interface for device code:
  + Drivers use a defined interface to kernel services (e.g. kmalloc, IRQ (interrupt-request) handler etc.)
  + Allows the Kernel to evolve without breaking existing drivers.
* Together both uniform interfaces avoid a lot of programming when implementing new interfaces.

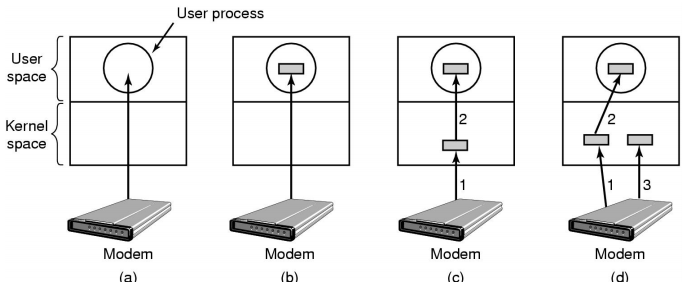
**I/O Buffering**

a) Un-buffered input

b) Buffering in user space

c) Single buffering in the kernel, followed by copying to user space.

d) Double buffering in the kernel.



(a) No Buffering

Process must read/write a device a byte/word at a time.

* Each individual system call adds significant overhead.
* Process must wait until each I/O is complete.
  + Blocking/interrupt/waking adds to overhead.
  + Many short runs of a process is inefficient (poor CPU cache temporal activity)

(b) User-Level Buffering

Process specifies a memory buffer that incoming data is placed in until it fills

* Filling of buffer can be done by interrupt service routine.
* Only a single system call and block/wakeup per per data buffer = much more efficient.

Issues with user-level buffering:

* What happens if the buffer is paged out to disk?
  + Could lose data while unavailable buffer is paged in.
  + Could lock buffer in memory (needed for DMA), however many processes doing I/O reduce RAM available for paging. Can cause a deadlock as RAM is a limited resource.
* Considering the write() case (can only be accessed one-at-a-time):
  + When is the buffer available for re-use?  
    Either the buffer must block the buffer until a potentially slow device drains the buffer
  + OR deal with asynchronous signals indicating buffer is drained.

(c) Single Buffering in Kernel, followed by copying to User-Space

OS assigns a buffer in Kernel’s memory, for an I/O request.

In a *Stream-Orientated* scenario:

* Single buffer is used a line at a time.
* User input from a terminal is one line at a time with carriage return signalling the end of a line.
* Output to the terminal is one line at a time.

In a *Block-Orientated* scenario:

* A block is copied to User-Space when needed.
* Another block is written into the buffer: read-ahead.

User-process can process one block of data while the next block is read in.

Swapping can occur since input is taking pace in system memory, not user memory.

OS keeps track of assignment of system buffers to user processes.