The OS is an **abstract machine**: hides hardware details, provide abstraction to programmers and applications.

*=> Service: File System, abstraction of named repositories of data | Benefit: apps are portable across diff hardware platforms.*

The OS is a **resource manager**: ensure no starvation, progress for all processes, allocation according to desired policy.

*=> Service: Scheduler, assign CPU execution time to processes/threads | Benefit: ensure system is responsive, parallelised.*

**THREADS & PROCESSES**

**Dispatcher/Scheduler** gives control of the CPU to a process that has been selected by the **Short-Term Scheduler.**

**Process Control Block (PCB)** **/ Context Block** stores all information needed about a process. Identified by a **Process ID (PID)**.

**Process Table** is a table of all the **PCBs** in a system. Facilitates Context Switching, Scheduling and other activities.

**Implementation of Threads & Processes**

Implementation at User-Level: User Thread-Control-Block + Ready Queue + Blocked Queue + Dispatcher

* Thread management (create(), exit(), yield(), wait()) are implemented in a runtime support library.

Implementation at Kernel-Level: Kernel Thread-Control-Block.

* Thread management (create(), wait(), exit()) are implemented as syscalls.
* TCB’s have a related PCB that the thread lives on.

Pros and Cons:

|  |  |  |
| --- | --- | --- |
|  | @USER-LEVEL | @KERNEL-LEVEL |
| PROS | * Context Switching + Threat Management is faster than @Kern (trap() to Kernel and back to User not required) * Programmer can choose the scheduling / Dispatcher algorithm for each application. * Can be implemented on any OS. * Can support huge numbers of threads. -> Uses application virtual memory | * **Pre-emptive Multithreading/Scheduling**: 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)   -> **Co-operative Multithreading/Scheduling**: 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)*  -> A single poorly implemented user-thread can monopolise the available CPU time.   * Does not take advantage of multiple CPU’s. * If a thread makes a blocking system call (a syscall which causes a thread to wait until the Syscall returns), the process and all internal threads will block i.e. a “blanket block” -> Can’t overlap I/O with computation. | * Thread creation, destruction, blocking and unblocking threads require Kernel entry and exit = more expensive operations than User-approach. |

**Context Switch** is the process of storing the state of a thread/process, so it can be restored and resume execution later.

* A Context Switch occurs when the OS is invoked on a Syscall, Exception, Interrupt.

Example Context Switch Scenario:

1. (USER MODE) SP is pointing to User 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 Program Counter and Stack Pointer).
3. ‘C’ code is executed to process the Exception/Syscall/Interrupt. “C activation stack” builds up.
4. The Kernel decides to perform a context switch.
   1. It chooses a Target Thread/Process + pushes remaining Kernel context onto the stack.
5. We save the current SP in the Process Control Block (or Thread Control Block) and load the SP of the Target Thread.
   1. We have now switched contexts.
6. Load the Target Thread’s previous context 🡪 return to C / User Mode.

**FILE MANAGEMENT & FILE SYSTEMS**

**Calculating Maximum File Size**

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

Direct Blocks = 12 **(given)**

Single Indirect Blocks = 1Kb/4 bytes = 256 **(Block Size / Block No. Size)**

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**

**Disk-Arm Scheduling Algorithms**

Time to read/write to a disk is determined by 3 factors: (1) Seek Time (2) Rotational Delay (3) Actual transfer time

#1 Random Order: Worst possible performance.

#2 First-In-First-Out (FIFO): process requests as they come.

* Fair - no starvation.
* Good for a few processes with clustered requests. Deteriorates to random if there are many processes.

#3 Shortest Seek Time First (SSTF): select requests that minimise seek time.

* Generally performs much better than FIFO.
* May lead to starvation

#4 Elevator Algorithm (SCAN): Move head in one direction – service requests in track order until last one, then reverse direction.

* Better than FIFO, usually worse than SSTF.
* Avoids starvation. Makes poor use of sequential reads. Inner tracks are serviced more frequently than outer tracks.

#5 Modified Elevator (Circular SCAN / C-SCAN): Elevator, but reads sectors in only one direction.

* When reaching last track, go back to first track non-stop. (seeking across the disk in one movement)
* Better locality on sequential fields. Better use of read-ahead cache on controller.

**File Allocation Methods**

A file is divided into blocks: what method is used to choose where to put the blocks on disk?

#1 Contiguous Allocation: In-order allocation

* Easy book-keeping (only need to track starting block + length of file)
* Increases performance for sequential operations
* Needs maximum size at allocation
* As files are deleted, free space becomes divided into many small chunks (External Fragmentation)

#2 Dynamic Allocation: Disk-space is allocated in portions as needed. Allocation is in fixed-sized blocks.

* NO external fragmentation
* Does NOT require pre-allocating disk-space
* Leads to partially-filled blocks (Internal Fragmentation)
* Complex metadata management (maintain a list of blocks for each file)

#3 Dynamic Allocation - Linked List: Each block contains a pointer to the next block in the chain. FREE BLOCKS are also linked.

* Best for sequential files.
* Poor for random access, blocks end up scattered across disk due to FREE BLOCK list eventually being randomised.
* Only single metadata entry per file.

#4 Dynamic Allocation - File Allocation Table: Keep a MAP of the entire File System in a separate table.

* A table entry contains a number of the next block of the file.
* The last block of a file and empty blocks are marked using reserved values.
* Table is stored on disk + replicated in memory.
* Random access is fast.
* Issues: Requires a lot of memory for large disks + Free-block lookup is slow.

#4 Dynamic Allocation – Inode based FS-structure (i-node = index node): The most popular File System structure today.

* Separate table / i-node for each file: each i-node stores attributes and disk-block location of the object’s data.
* Only keep a table for open files in memory = Fast Random Access.
* Inode implementation issue: allocated dynamically, hence free-space management is required for inodes.

**MEMORY MANAGEMENT (PRE-VIRTUAL MEMORY)**

**Swapping**: Transferring an entire process address space to/from the swap device (typically to/from Disk)

**Paging**: Copying to/from pages of the address space = finer grain. **Overlay**: Keep in memory only instructions/data needed.

**Mono-Programming**: Simple memory management without swapping/paging. Running 1 program at a time. (embedded device)

**Dynamic Partitioning Strategies**:

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

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

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

* Intent is to 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 lead to lots of unusable holes at the front = External Fragmentation

#3 Next-Fit Approach: Scan the list from the point in the list where the last request succeeded.

* Spreads allocation more equally throughout memory, but often allocates a block at the end where the largest block is.
* The largest free block is eventually broken into smaller blocks, unable to service larger requests.

#4 Best-Fit Approach: Chooses the block that is closest in size to the request.

* Poor performer, has to search the complete list. Smallest block fit chosen = least amount of External Fragmentation

#5 Worst-Fit Approach: Chooses the block that is largest in size (worst fit)

* Idea is to leave a usable fragment left over, which it FAILS to do properly.
* Poor performer, has to do more work to search the complete list.

**CONCURRENCY & SYNCH PRIMITIVES**

**Race Condition**: Occurs when >= 2 threads access shared data and try to change it at the same time.

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

* Solution to C.R must satisfy: **Mutual Exclusion | Progress | Bounded Waiting**

**Bounded Waiting** prevents **Starvation**: Starvation occurs when a process cannot acquire a resource because the scheduler always allocates resources to other processes, denying the starved process from progressing.

* Solution to Starvation: Implementing a FIFO policy (queues).

**Semaphore:** used to control access to a shared resource.

* 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 -> ready queue.

**Mutex**: similar to a Semaphore (sem val = 1) except only the threads that hold the Mutex may unlock it.

**Monitor:** a group of procedures, variables etc. grouped into a module, where only one thread can be inside the monitor.

* When a thread calls a monitor procedure with an existing thread inside, it is queued and sleeps until the current thread exits the monitor.
* Consists of a MUTEX + CONDITIONAL VARIABLE.

**Conditional Variables**: provides a queue for threads waiting for a resource, to wait for a condition to occur.

* While a thread is waiting on a conditional 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.

**Concurrency/Synch Problems and Solutions**

Producer-Consumer Problem (Bounded-Buffer Problem)

Producer + Consumer who shares a common, fixed-size buffer. Producer adds items. Consumer consumes items.

* PROBLEM: Make sure the producer doesn’t add to a full-buffer + consumer doesn’t consume from an empty buffer.
* SOLUTION: **2 semaphores (num\_empty | num\_full)** + **1 shared mutex**

Dining-Philosopher’s Problem

K philosophers seated in a circle. Philosopher may eat with 2 chopsticks. Chopstick can only be used by 1 person at a time.

* PROBLEM: Make sure Deadlock doesn’t occur.
* SOLUTION #1: **Semaphore Array** + **1 Mutex**. One semaphore per chopstick to control behaviour of each philosopher.
* SOLUTION #2: **Resource ordering**. Anytime a process requests a resource, it has to request it in a global specified order.

Readers and Writers Problem (models access to database)

Possible to have more than one concurrent reader. However, writers must have exclusive access when writing (0 reader/writers)

* PROBLEM: Make sure a race condition does not occur / threads have proper synchronisation
* SOLUTION: **1 Semaphore** + **1 shared Mutex**. Semaphore to control access to Read\_Ctr, Mutex to control access to DB.

**DEADLOCKS**

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

**Four conditions of Deadlock:**

1. **MUTUAL EXCLUSION**: Several processes cannot simultaneously share a single resource.  
   (read-only resources do not require this condition therefore can’t be involved in a deadlock)
2. **HOLD AND WAIT**: Hold the resource I have + wait for the resource that I don’t have yet.
3. **NO PRE-EMPTION**: Resources cannot be taken away pre-emptively by another process or by the OS.
4. **CIRCULAR WAIT**: A chain of processes where each process is waiting for a resource that is held by another process.

**Strategies for dealing with Deadlock:**

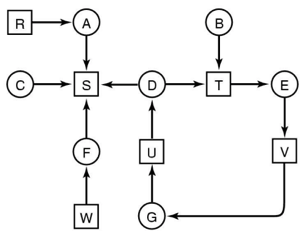
**THE OSTRITCH ALGORITHM**: ignoring problems on the basis that they are extremely rare to occur.

* Reasonable if Deadlocks rarely occur and the cost of prevention is high. (Convenience vs. Correctness trade-off)

**DEADLOCK PREVENTION**: prevent one of the four Deadlock conditions.

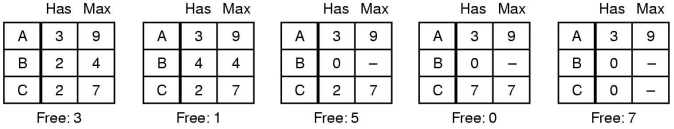
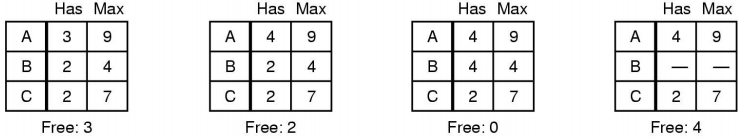
* #1 Mutual Exclusion: NOT FEASIBLE. Some devices/resources cannot be made shareable e.g. Printer.
* #2 Hold-and-Wait: FEASIBLE. Require processes to request ALL resources before starting execution.
  + Not always possible to know all the required resources at the start of execution.
  + Variation of the strategy: process gives up all its resources if it blocks another process, then immediately request all resources again.
  + ^**LiveLock** can occur: set of processes that aren’t blocked, run regularly but never make any progress.
* #3 No Pre-emption: NOT FEASIBLE. Can’t take resources away from a printer halfway through printing.
* #4 Circular Wait Condition: FEASIBLE/BEST. **Resource Ordering** is a common technique used in practice.

*Anytime a process requests a resource, it has to request it in a global (applying to all processes) specified order.*

**DEADLOCK DETECTION & RECOVERY**: let deadlocks happen and come up with a recovery mechanism.

* Resources w/ single unit: look for cycles within the graph, denoting a deadlock. --------------------->
* Resources w/ multiple units: 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.
* Deadlock recovery: recovery is performed through killing one of the processes in the deadlock.

**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 a deadlock.
* Unsafe State: Processes are not necessarily deadlocked, but we cannot guarantee that they will not deadlock/complete.
* **Banker’s Algorithm**: **Current Allocated (C) + Resources Available (A) = Resources in Existence (E)**
  + Keep the bank in a “SAFE STATE” where all customers can request to borrow up to their limit simultaneously.
  + Customers wishing to borrow such that the bank would enter an UNSAFE STATE must wait until someone repays their loan such that the transaction becomes safe.
  + Banker’s Algo is not commonly used as it is difficult to determine the amount of resources required in advance.
  + Algorithm:

1. Calculate Available (A) = Existence (E) – Current Allocated (C)
2. Find a row/process in the Requested matrix where Requested <= Available
3. IF row/process exists, completion is possible | ELSE deadlock is possible, implying INITIAL STATE = UNSAFE.
4. Selected row/process acquires available resources -> execute -> terminate -> return resources back to Available (A)
5. Repeat steps until all processes have successfully reached termination, implying INITIAL STATE = SAFE.

**MEMORY MANAGEMENT & VIRTUAL MEMORY (Page-Based)**

**Internal Fragmentation vs. External Fragmentation**

Internal Fragmentation is when address space INSIDE an allocated region is wasted, due to fixed-size block allocation.

External Fragmentation is when address space OUTSIDE an allocated region is wasted, due to dynamic allocation.

* Solved by Compaction, Paging or Segmentation (user-view of memory, separating memory into logic units)
* **Compaction**: reducing 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.
  + Requires *base* and *limit* registers: special hardware registers define the logical address space of a process.

../../../../Desktop/Screen%20Shot%202018-06-17%20at%204.12.53%20**Effective Memory Access Time Calculation**

Example #1: Cache Memory vs. Main Memory

Cache memory access time = 1ns | Main memory access time = 10ns | Hit Rate = 95%

* **(0.95 \* 1ns) + ((1 – 0.95) \* (10ns))**

Example #2: TLB, Page Table and Disk-Read scenario

Page Table access time = 100ns\*2 (per memory reference) | Paging Disk = 5ms per page (incl. access/transfer)

TLB Hit Rate = 99% | TLB Miss Rate = 1% | Page Fault Rate = 0.0005%

* **(0.99 \* 100ns) + ((0.01 – 0.0005) \* 2 \* 100ns) + (0.0005 \* (3 \* 100ns + 5ms))**

*3 memory references \* 100ns => updatePT() + access PT + updateTLB()*

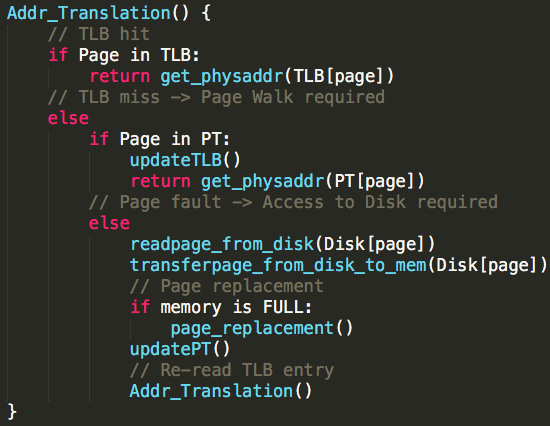
**Page Size / Page Table Calculation**

Assume 32bit virtual address space (2^32 = address values), mapped to 4kB size pages.

* PAGE SIZE IN BITS: 4kB page size = 4,000bytes = **~2^12 bits per page**
* NUMBER OF PAGES: 2^32 / 2^12 = **2^20 no. of pages & no. frames (page size = frame size)**
* PAGE TABLE ENTRY 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**

**Virtual to Physical Address Translation (TLB 🡪 PAGE TABLE 🡪 DISK process)**



**Page Tables: Mapping Virtual Memory to Physical Memory (virtual page number to page frame number)**

Two-Level Page Table: Index=**MasterVPN+RootPtr: [ 2ndRootPtr ] 🡪** Index=**2ndRootPtr+2ndVPN: [ PFN ]**

*Virtual Address: [ MASTER VPN | 2nd LEVEL VPN | OFFSET ]*

* Master PT Entry = Master Page No. + Root Page Table Ptr
* 2nd Level PT Entry (Page Frame Number) = Master PT Entry + 2nd Level Page No
* Physical Address = 2nd Level PT Entry + OFFSET

Inverted Page Table: Index=**PFN: [ PID | VPN | CTRL | NEXT ]**

*Virtual Address: [ PID | VPN | OFFSET ]*

* HashVal = Hash(VPN) 🡪 HashTable[HashVal] 🡪 Inverted PT Index (Frame Number) 🡪 IPT[FrameNumber]
* Match PID & VPN in Inverted PT Entry
* *If match:* use IPT index (Frame Number) | *If no match:* goto next entry in chain | *if NULL chain*: page fault

Hashed Page Table: Index=**HASH: [ PID | VPN | PF` | CTRL | NEXT ]**

*Virtual Address: [ PID | VPN | OFFSET ]*

* Key = Hash(VPN) 🡪 HPT[Key]
* Match PID & VPN in Hashed PT Entry.
* *If match:* use PFN value + OFFSET | *If no match:* goto next entry in chain | *if NULL chain:* page fault

**Locality, Working Set, Thrashing**

Principal of Locality: Programs would typically access data and storage that they have used recently, so we can reasonably predict what data/location a program will access in the near future based on recent accesses.

* **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: Are the pages required by an application in a specified time window.

* If window too small => may not encompass the entire location required.
* If window too large => may encompass several locations.
* If window approaches infinity => will encompass the entire program.

Thrashing: If a process’s Working Set no longer fits in available RAM, repetitive swapping may occur and page fault rate becomes high, leading to high CPU utilisation and crashing the system. RECOVERY FROM THRASHING:

1. Suspend a few processes to reduce the degree of multiple running processes.
2. Migrate pages of suspending processes to a backing store.
3. More physical memory becomes available -> less faults, faster progress for running processes
4. Resume suspended processes later when memory pressure eases.

**Page Replacement Policies**

Optimal Replacement: Remove the page that won’t be used for the longest time.

* Impossible to implement, as it is difficult to predict how long until a page will be used.

FIFO Replacement: Remove the oldest page. Easy to implement, however age of a page is not necessarily related to usage.

Least Recently Used (LRU): Remove the least recently used page.

* Assumes that a page has not been referenced for a long time + unlikely to be referenced in the near future.
* Implementation requires a TIME STAMP to be kept for each page, UPDATED ON EACH REFERENCE.
* **Most practical algorithms are approximations of LRU**. LRU itself is impossible to implement efficiently.

Clock Page Replacement: Remove the 1st page with a 0 reference bit in a circular list of frames.

* Maintains a REFERENCE BIT for each frame, set to 1 when a page is used.
* While scanning for a victim, reset BIT for frames with BIT=1.
* How do we know when a page is referenced?
  + When a page is mapped to a Page Table Entry (valid bit set), set the reference bit.
  + When resetting the reference bit, invalidate the PTE.

**INPUT / OUTPUT MANAGEMENT**

**I/O challenges**: Challenge with I/O management is having a **UNIFORM** and **EFFICIENT** approach.

Uniform: Handle all I/O devices the same way.

* Problem is the diversity of I/O devices, different access methods (random vs. stream) and different data rates.
* Hide most of the details of I/O in lower-level routines so processes/apps only see read() write() open() close() etc.

Efficient: Most I/O devices are slow compared to RAM and the CPU.

* Multi-programming allows for processes to be waiting on I/O while another process executes.
* GOAL: optimise I/O efficiency, especially Disk and Networking I/O.

**I/O device handling considerations**:

Complexity of Control: mix of simple and complex controls

Unit of Transfer: stream of bytes for a terminal / larger blocks for a disk

Data Representation: different encoding streams

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.

**Types of I/O Interactions (example: CPU issues READ command to I/O)**

Programmed I/O(or Polling/Busy Waiting): I/O controller performs the action, not the processor.

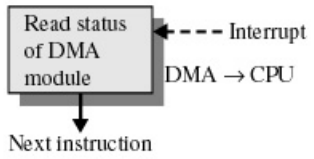
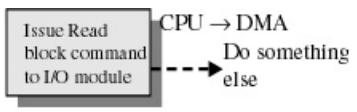
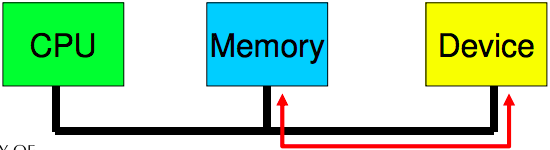
* Processor checks I/O status continuously to see if it is READY, NOT READY or ERROR CONDITION.
* This continuous polling wastes CPU cycles. No interrupts occur.

Interrupt-Driven I/O: Processor continues to do work and is interrupted when I/O controller is ready to exchange data.

* This consumes a lot of CPU time as every word read/written will interrupt the CPU.

Direct Memory Access: Transfers blocks of data directly between the Device and Memory, so the CPU is not needed for copying.

* An interrupt is sent when the task is complete. CPU is only involved at the beginning/end of transfer.



Process to perform Direct Memory Access transfer of data:

1. CPU tells Device driver 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.

DMA PROS: Reduces no. of interrupts = less expensive context switches or kernel entry-exits.

DMA CONS: Requires contiguous regions (buffers)

**I/O Software Layers: Interrupt Handlers**

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.

Interrupt handlers need to finish quickly to allow other important interrupts to occur, but may have a lengthy task to perform. This lengthy task can be postponed at a later time, so there needs to be a way to store the context for later use.

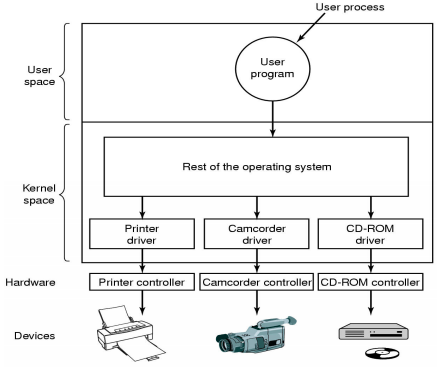
* Solved by dividing interrupts into two parts: **Top & Bottom Halves**
* Top Half: the routine that responds to the interrupt. (interrupts disabled)
  + Saves device data to a device-specified buffer + schedules the Bottom Half and Exits.
* Bottom Half: the routine scheduled by the Top-Half to be executed later (interrupts re-enabled)
  + Performs all other work that is required e.g. awakening process, starting up another I/O operation etc.
  + Work is deferred to In-Kernel threads: low latency + low blocking operations.
* **This method allows the Top-Half to service a new interrupt while the Bottom-Half is performing the remaining tasks.**

**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.

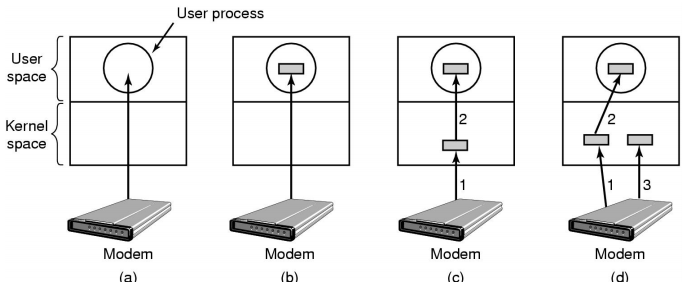
**I/O Buffering**

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.

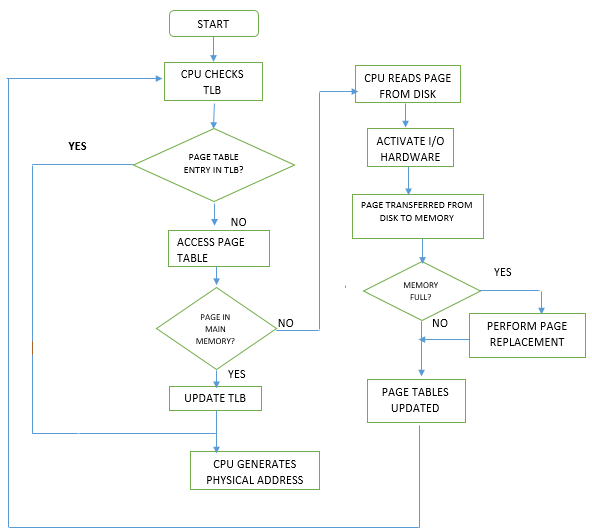
(d) Double Buffering: Two system buffers

A process can transfer data to a buffer while the OS fills / moves data to the other buffer.

Computation and Memory copy can be done in parallel with transfer speed.

Double Buffering may be insufficient for bursty traffic. May want to read-ahead more than a single block for a Disk.

**Translation Look-aside Buffer FLOWCHART**



**Two-Level Page Table**

