**OS**

**Learning Outcomes:**

* High-level understand what is an operating system and the role it plays
* A high-level understanding of the structure of operating systems, applications, and the relationship between them.
* An understanding of fundamental concepts of processes and threads
* Understand concurrency is an issue in operating systems and multithreaded applications
* Know the concept of a critical region.
* Understand how mutual exclusion of critical regions can be used to solve concurrency issues
* Including how mutual exclusion can be implemented correctly and efficiently.
* Be able to identify and solve a producer consumer bounded buffer problem.
* Understand and apply standard synchronisation primitives to solve synchronisation problems.
* Understand what deadlock is and how it can occur when giving mutually exclusive access to multiple resources.
* Understand several approaches to mitigating the issue of deadlock in operating systems.
* Including deadlock prevention, detection and recovery, and deadlock avoidance.
* A basic understanding of the MIPS R3000 assembly and compiler generated code.
* An understanding of the typical implementation strategies of processes and threads
* Including an appreciation of the trade-offs between the implementation approaches
* Kernel-threads versus user-level threads
* A detailed understanding of “context switching”
* A high-level understanding of System Call interface
* Mostly from the user’s perspective•From textbook (section 1.6)
* Understanding of how the application-kernel boundary is crossed with system calls in general
* Including an appreciation of the relationship between a case study (OS/161 system call handling) and the general case.
* Exposure architectural details of the MIPS R3000
* Detailed understanding of the of exception handling mechanism
* From “Hardware Guide” on class web site
* Understand the concepts of memory hierarchy and caching, and how they affect performance
* Appreciate the need for memory management in operating systems, understand the limits of fixed memory allocation schemes.
* Understand fragmentation in dynamic memory allocation, and understand basic dynamic allocation approaches.
* Understand how program memory addresses relate to physical memory addresses, memory management in base-limit machines, and swapping
* An overview of virtual memory management.
* An understanding of page-based virtual memory in depth.– Including the R3000’s support for virtual memory.
* An understanding of TLB refill:– in general, – and as implemented on the R3000
  + An understanding of demand-paged virtual memory in depth, including:– Locality and working sets– Page replacement algorithms– Thrashing
* An understanding of the structure and limits of multiprocessor hardware.
* An appreciation of approaches to operating system support for multiprocessor machines.
* An understanding of issues surrounding and approaches to construction of multiprocessor synchronisation primitives.
* Understand the role of the scheduler, and how its behaviour influences the performance of the system.
  + Know the difference between I/O-bound and CPU-bound tasks, and how they relate to scheduling
* A high-level understanding of the properties of a variety of I/O devices.
  + An understanding of methods of interacting with I/O devices.

**What is the role of the OS?**

1. Provides high level abstractions, hides detail, and extends basic hardware
   1. Raises low level functions to more program-friendly, higher level abstractions
2. Resource management
   1. No starvation, allocation to some desired policy
   2. Multiplexing 🡪 Sharing resources in time and in space

**How is it implemented:**

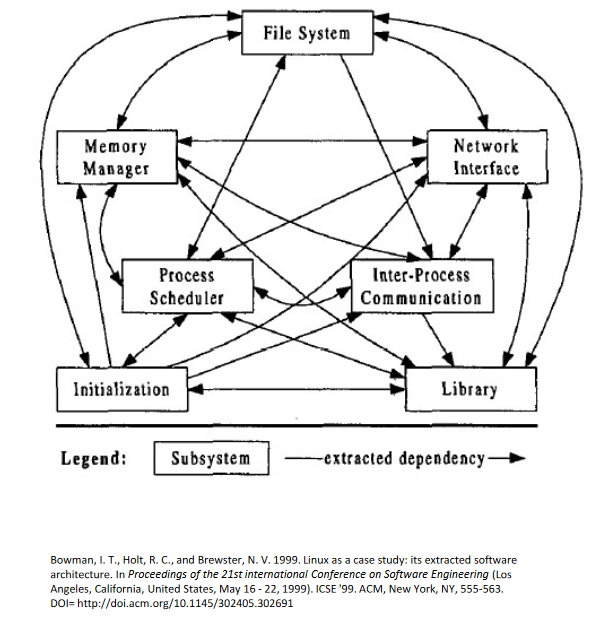
****

Figure Abstraction of The Linux Operating System Structure

* OS is the software that runs in the privileged mode of a microprocessor
  + Privileged mode has access to the entire architecture compared to user mode (applications)
  + OS software and application software is fundamentally the same
    - Main difference is determined by the mode the chip is in when it is running the instructions (privileged vs user)

OS Kernel: Portion of the OS that is running in privileged mode

* OS interacts via load and store instructions to all memory, cpu and device registers and interrupts

Assume applications are malicious: OS should not allow applications to bypass/interfere with itself

OS relinquishes control of the processor to execute applications, re-establishes control after:

1. System calls
2. Interrupts (e.g. timer interrupts)

**Internal Structure**

Process: Execution of an individual program

* OS attributes resources to a process for a program execution, encompassing one or more threads
  + OS keeps track of processes within a process table
* Effectively a container for threads
  + Not to be confused with a program: which is just the algorithm expressed in some language/notation.
  + A process is an activity of some kind: it includes the program, I/O and state
* A suspended process contains it’s address space (core image) and it’s table entry (what is needed to restart the process later)

Thread: Unit of execution

* A sequence of executions that can be traced. Belongs to a process
* Why use threads?
  + Less resources than a complete process
    - Easier to create and destroy
  + Simpler than a state machine
  + Good for performance when you overlap I/O with computation
    - E.G. Multiple activities may block, threads add quasi-parallelism and allow processes to run even when individual threads are blocked by I/O
  + Takes advantage of multiprocessing with multiple threads of control (>1 cpu)
  + Sharing address space and data means threads can work on the same address space simultaneously

**The Process Model:**

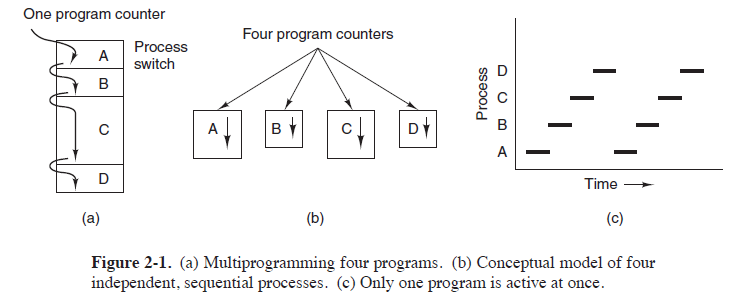


Figure Visualisation of Process Execution

* Multiprocessing: An OS will have a scheduler that switches between processes . Each process is active for a short period of time but the process itself works as if it was continuously active
* Process information is stored in a process control block (PCB)
  + PCBs form a process table
  + Each process has its own virtual cpu

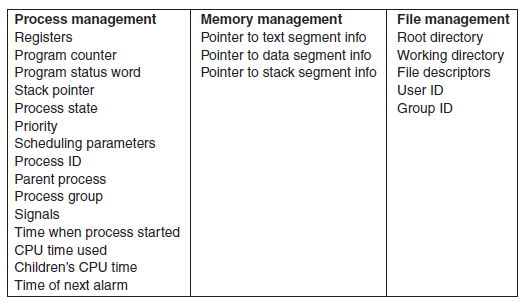


Figure Example Process Control Block

* Three State Process Model:
  + Ready: Process can run and is waiting for the process scheduler to run it
  + Running: Process is being run on the cpu
  + Blocked: Process is blocked (I/O Wait)
    - Something with the semantics where we expect the result to exist when the function returns. Thus the program waits until the blocked action is finished

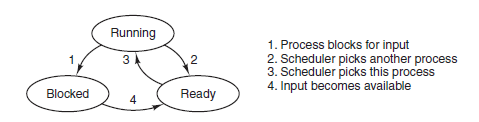


Figure Possible transitions between Process States

Scheduler / Dispatcher: Chooses a *Ready* process to run

* Typically uses a queue
* Can dynamically create separate queues for different events to address blocked processes

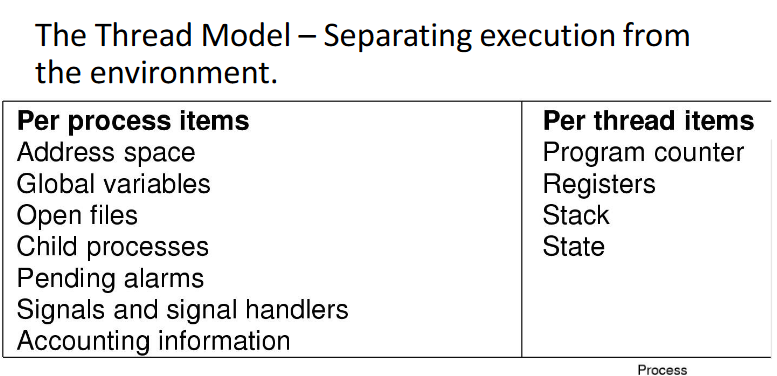


Figure Thread Model: What Processes and Threads keep track of

**The Thread Model:**

* Local variables are private on the local stack per thread
* Global variables are shared 🡪 Concurrency becomes an issue!
* Dynamically allocated memory can be global or local
  + Based on the pointer (local or global)

**Event Based Model** / **Finite-State Machine Model**: Reacts to input events, and does bookkeeping to keep track of event dependencies (allows us to attribute finished actions to the causal input event)

* Not a sequential model; The state of computation must be explicitly saved and restored in the table every time the cpu switches from one request to another.
* Effectively causes the system to restore the state as if the system were continuing immediately from the reply.
  + Bookkeeping arises because the computer needs to keep track of the previous state in order to continue the execution of code after each new event
  + This bookkeeping is explicitly managed by the program
* Asynchronous system calls: Model using non-blocking actions. Program continues execution (returns immediately but blocked action may not be ready). When it reads a finish signal for the “blocked” action, the program returns to the logic of the original request
  + Essentially bypasses blocked action to do something else until blocked result is returned
  + Works because we are keeping track of the outstanding requests

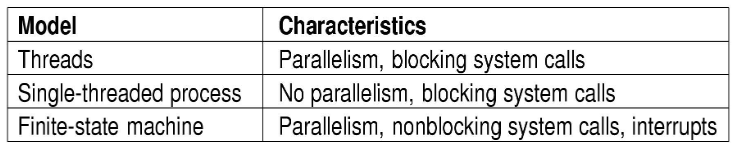


Figure Different Types of Thread Usage Models



Figure Difference in Computation State between Thread and Event based models (Example: Flipping Burgers)

**InterProcess Communication (IPC):** Mechanisms an OS provides to allow processes to manage shared data. The shared data / region of code where shared resources are accessed is called the **Critical Region**

* **Race Conditions** are where the system’s behaviour is dependent on the timing of processes/threads. This can result in bugs when two processes read and write to the same shared memory, thus the arbitrary timing of the scheduling may cause a bug in the output of the system

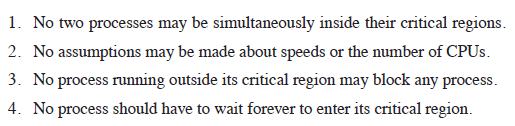


Figure Conditions for a solution to Race Conditions w.r.t. IPC

* To avoid race conditions, processes must be **mutually exclusive** when accessing critical regions

**Mutual Exclusion in Critical Regions**: Solutions (based on Fig 8 conditions)

* Taking turns
  + PROS:
    1. Works due to strict alternation
  + CONS:
    1. Busy waiting
    2. Process must wait it’s turn even while other process is doing something else, or using the critical section at different rates
* Disabling interrupts
  + Not appropriate
* Hardware support
  + **Test-and-set** instructions are atomically (indivisibly) executed
    - Essentially uses a lock variable but hardware ensures that the test-and-set instruction for the lock variable is always executed as a block
    - Can keep separate locks for different file or I/O.
      * E.G. Network lock, file lock can be parallelised
      * Test-and-set instructions keep testing until the related lock is available to set (spin lock)
    - PROS:
      1. Simple
      2. Available at user-level
    - CONS:
      1. Busy waits / spin lock
         * Consumes CPU
      2. Possibility of starvation (waiting queues by scheduler)
* The **busy-wait problem:** Looping for test (checking a condition) is unproductive. Thus we introduce sleep/wakeup primitives
  + Sleep replaces busy waiting when blocked. After event, wakeup is called to unblock the sleeping process
  + Waking ready/running processes has no effect

Dining Philosophers Problem: We want to avoid deadlock by allocating and blocking resources. Models processes that are competing for exclusive resources

Readers and Writers: We want to let anybody read the data but we can’t have anybody reading at the same time we write to the data. Many simultaneous processes want to access a database

Resources in computing systems:

* Preemptable resources: Can be taken away from a process with no ill effects
* Non-preemptable resources: Will cause the process to fail if taken

**Deadlock**: The act of requesting an unavailable resource may cause processes to wait. If processes wait for resources in a cycle where each resource is taken and access is dependant on the resource holder, the resultant wait is deadlock

* Occurs when processes are granted exclusive access to devices, locks, tables, etc.
* Formal definition: A set of processes is deadlocked if each process in the set is waiting for an event *that only another process in the set can cause*

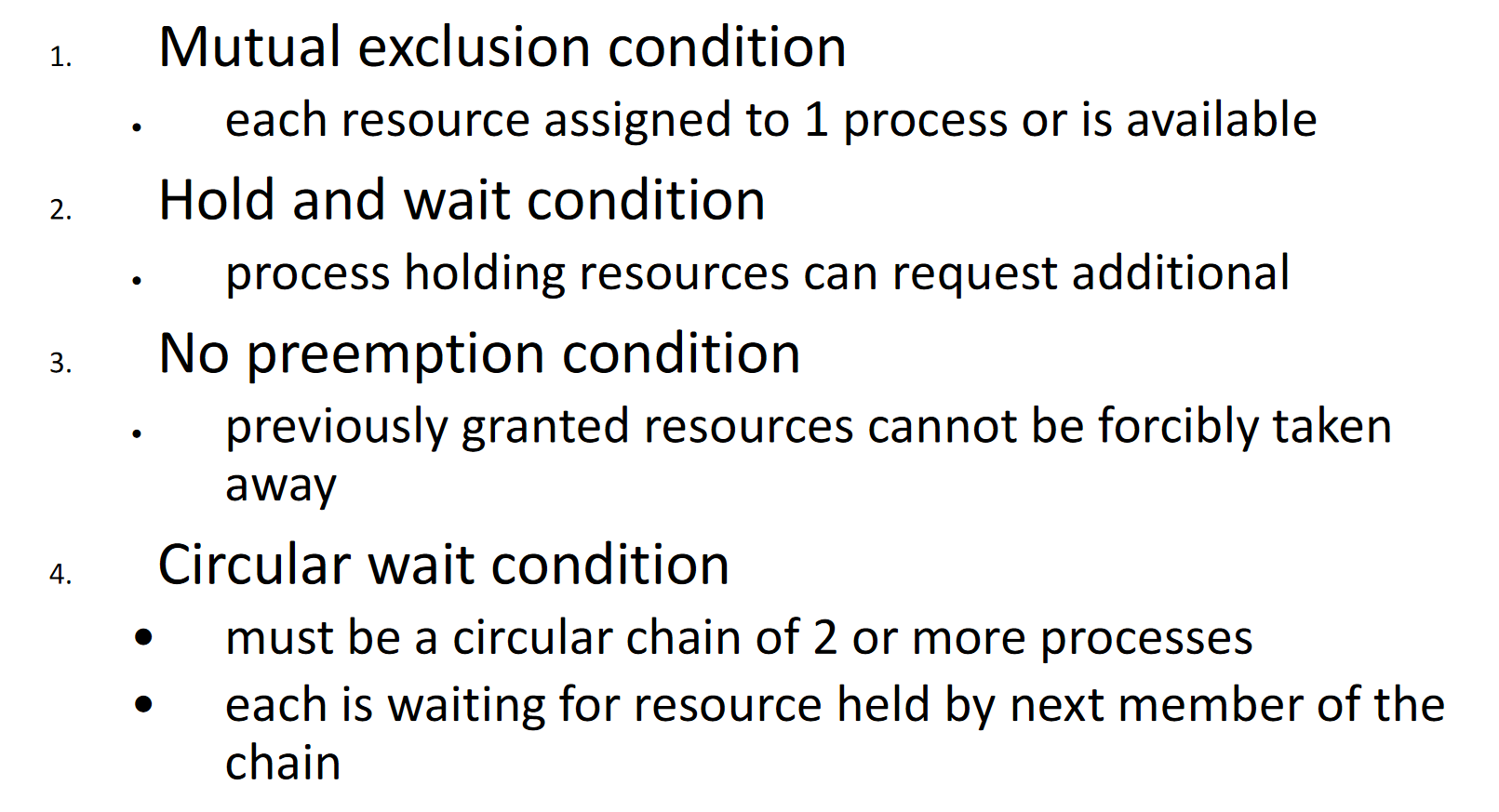


Figure Conditions for Deadlock to Occur

**Deadlock Strategies**:

* Deadlock Prevention: Create resource allocation rules to prevent one of the four deadlock conditions from happening
  + Can’t prevent mutual exclusion or no pre-emption
  + Hold and wait: 2 Methods, both have significant disadvantages
    - Require process to request resources beforehand
      * Not feasible on all systems
    - give back resources and repeat requests for resources
      * can result in livelock
  + **Circular wait**: Quantize labels (resource ordering) for critical regions. Thus, when acquiring two resources, you must acquire the resources in order.
    - Removes cycles in the resource wait dependencies, thus deadlock is impossible
    - Lock ordering needs to be enforced to avoid deadlock
* Detection and Recovery: Multiple resource instances. Matrices for resource allocations
  + Significant computation overhead

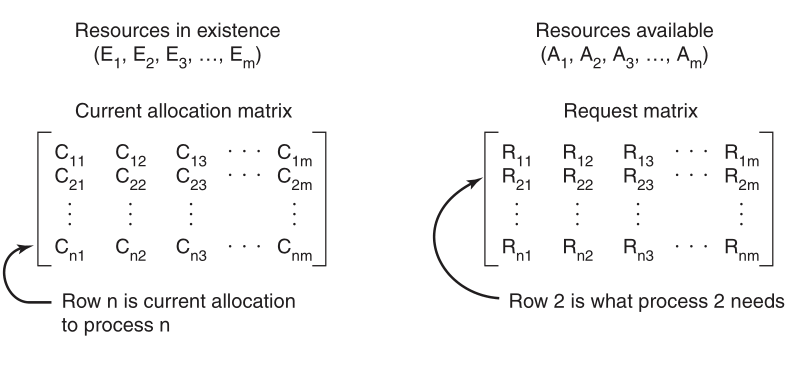


Figure Deadlock Detection Data Structures

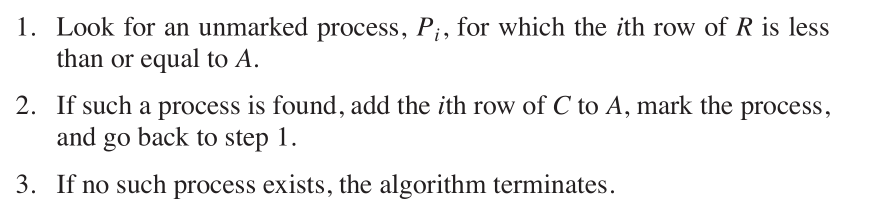


Figure Deadlock Detection for multiple instances of resources

* + How to recover from deadlock?
    - Not every resource is non-preemptable
    - Rollback? Checkpointing at regular intervals and reset to state. No guarantee it won’t deadlock again
    - Kill a process. Crude but simplest way to break a deadlock.. most people hate this
* Deadlock Avoidance: Different to prevention. Instead of detecting deadlock, can we simply avoid it?
  + Allocating resources such that deadlock doesn’t occur
  + Only practical for systems where the maximum quantities of needed resources are known
  + Banker’s Algorithm: Simulate the allocation of predetermined maximum possible amounts of all resources, then make a safe-state check to test for possible deadlock conditions for all other pending activities, before deciding which allocation should continue
    - Not commonly used in practice due to difficulty in knowing resource allocation in advance

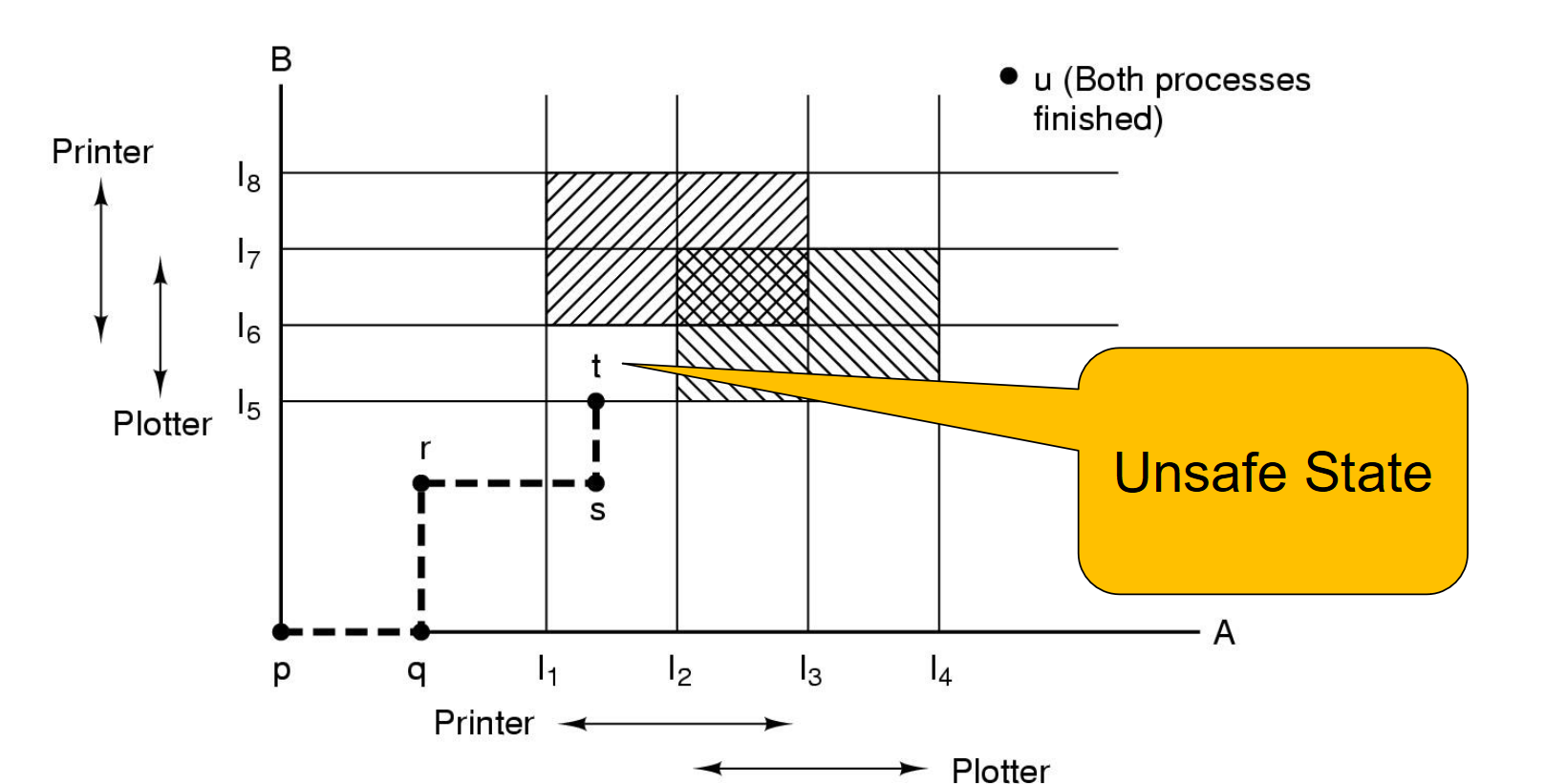


Figure Banker's Algorithm Motivation: Check resource allocation beforehand to avoid unsafe states where deadlock is inevitable

* Ostrich Algorithm: Take the engineering approach. Ignore it if it’s too improbable or not worth the cost to fix

Starvation: A process never receives the resource it is waiting for despite the resource repeatedly becoming free. The resource is always allocated to another process

* Solutions?
  + First come first serve policy?

Thread Implementation:

* IEEE standard: Pthreads (POSIX)
* Two main places to implement threads:
  + User space
  + Kernel space

**User-space threads**: Threads packaged entirely in user space

* Threads implemented by a library
  + Threads run on top of a run-time system (collection of procedures that manage threads)
  + Threads stored in a thread table by the process
  + If the OS supports exchange register instruction, there is a significant performance benefit from not trapping to the kernel to switch threads
* Can be implemented on an OS that does not support threads
* *Thread\_yield()­* can call the thread scheduler to pick another thread
  + No context switch required
  + No trapping required
  + Memory cache doesn’t need to be flushed
* Allows each thread to call each scheduler through *thread\_yield()*
* Problems with user-space threads:
  + Blocking system calls: A system call can’t block because that would block all threads in user space
  + May have to write a wrapper around system calls
    - See ‘select’ in UNIX
  + Needs to keep track of page faults
  + Doesn’t take advantage of multiple CPUs
    - The CPU sees the user-thread model as one process
  + No clock interrupts, so threads have to yield to other threads
  + Threads are most useful when they can be used to block, such as in a web server.

**Kernel-space thread implementation**:

* Thread information is a subset of process table, which is maintained by the kernel
* Calls that block are implemented as syscalls
* Pre-emptive multithreading: Timer interrupts are used to interleave execution of threads for concurrency
  + Abstracts thread\_yield() from the programmer
* Problems with kernel-space threads:
  + Cost of syscall is substantial
  + Signals are sent to processes. Which threads should handle signals?
  + How should forking work with multiple threads?

Context Switch: Saving and restoring of the state associated with a thread or process

System Calls: Special functions that provide a controlled entry into the system kernel

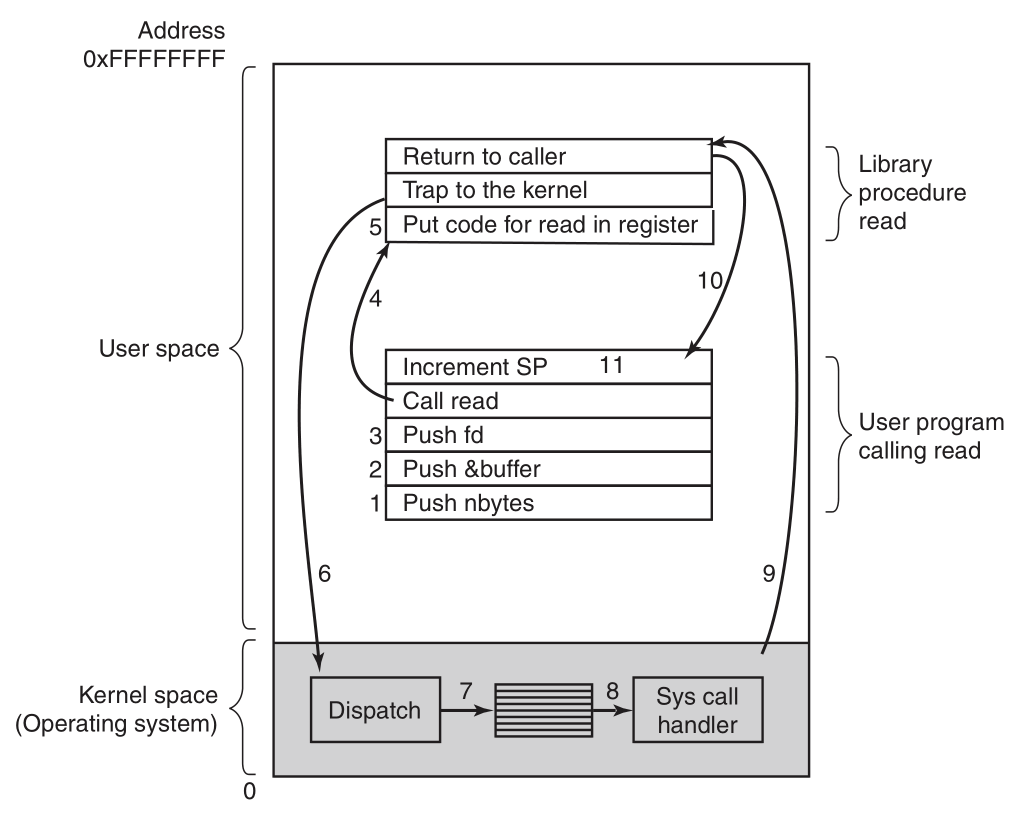


Figure 11 Steps for making the read syscall

* UNIX processes are divided into three memory segments:
  + Text: program code
  + Data: variables, grows downward
  + Stack: frames of memory, grows upwards
* System calls restrict possible entry points to the OS to secure locations
  + Guaranteed checks on syscalls

Privileged-mode Operation:

* The accessibility of addresses within an address space changes depending on operating mode: kernel vs user mode
* OS and application memory is kept separate

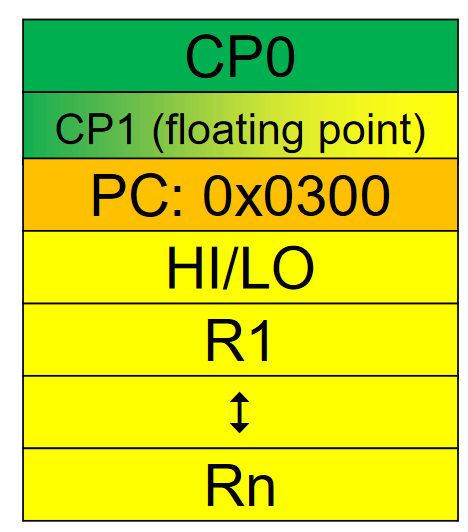


Figure MIPS Coprocessor 0

CP0: Sequence of registers used to manage the machine. Can only manipulate from the privileged mode of the architecture

CP0 Registers: There are a lot of c0 registers. Some examples below

* C0\_cause: cause of last exception
* C0\_status: current CPU status
  + Bit shift by 2 for current, past, old bits (bits 0-5 in the c0\_status register figure)
* C0\_epc: address of the instruction that caused the exception. Indirectly saves pc



Figure c0\_status register

**Interrupt vs Exception**:

* Exception: Anything that causes a transition between user and kernel mode
  + Syscall, hardware primitives that transition to kernel mode are both exceptions
  + Exception handler is hard coded in some memory location which the OS jumps to
  + KU/IE set based on mode
  + Entering exception: bit shifts c0\_status (KU/IE current 🡪 KU/IE past). Move to exception frame. Save address to c0\_epc
  + Returning from exception: Loads address from c0\_epc to register 27 (register for OS use). Bit shifts c0\_status back (KU/IE past 🡪 KU/IE current). Returns to state from which an exception was called
* Interrupt: Can cause exception

Exception walkthrough:

* How to return from an exception:

File: Address abstraction for the disk

* File extension: Part of the file name following a period, denotes something about the file
* Unstructured sequence of bytes: Type of file structure where the user-level programs impose meaning on the file
  + Structure of unix, windows
* Types of file access:
  + Sequential: Used in magnetic disks
  + Random Access: Accessed in any order with one of two methods: (a) seek or (b) from the start
* Attributes: Extra data / metadata the OS stores about the file.

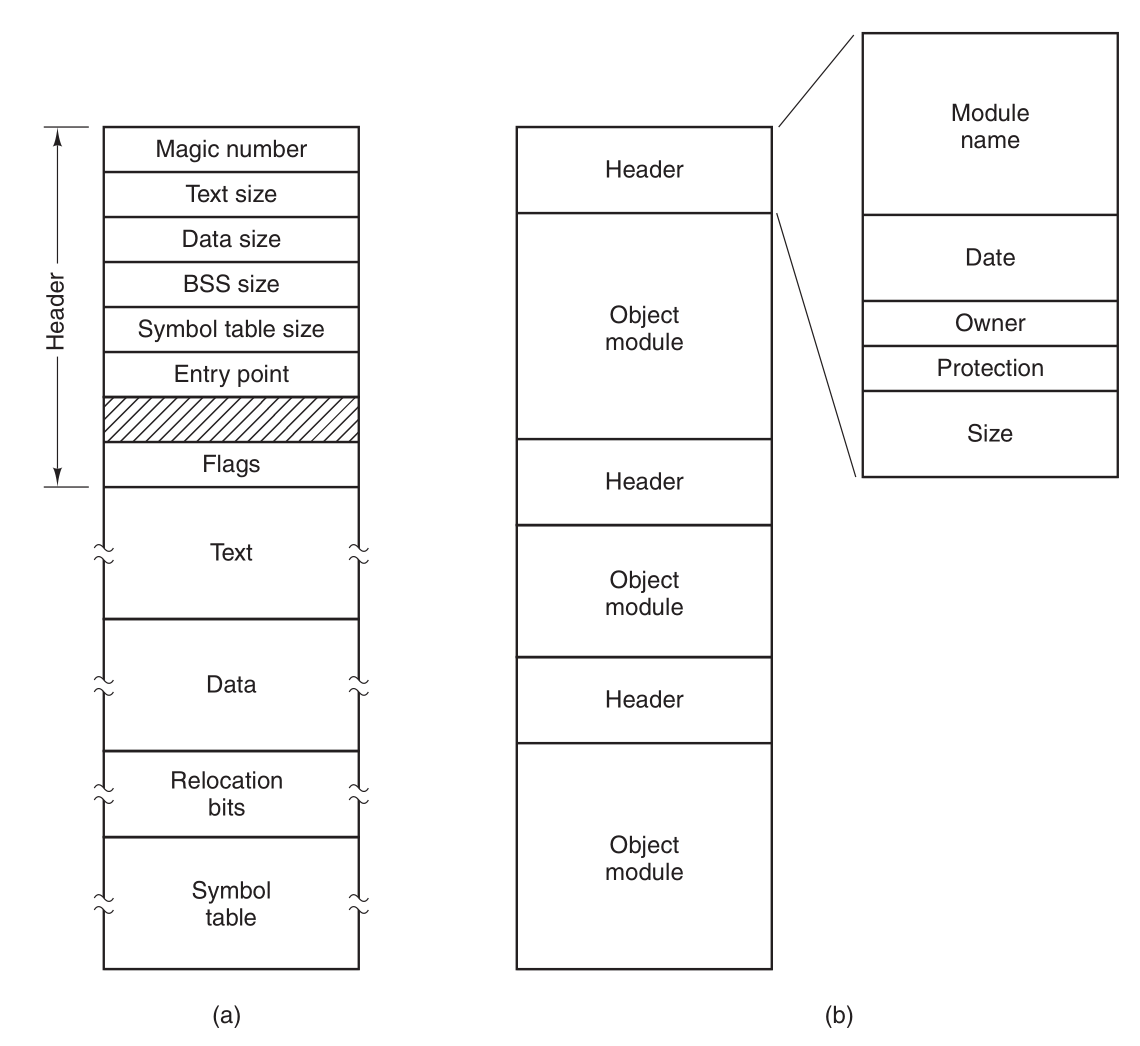


Figure (a) Executable File (b) An Archive

Directories: System files for maintaining the structure of the file system

Character special file: Used to model serial I/O devices

Block special file: Models disks

Directories: Container for objects

* Relative path name: path to a file from the working directory
* Each process has its own working directory.

File System Implementation:

* File systems are stored on disks. Disks are divided into one or more partitions, with independent file systems on each partition
  + Sector 0: Master Boot Record (MBR). Used to boot the computer
    - MBR locates first block of memory in the active partition: the boot block
* One active partition at a time

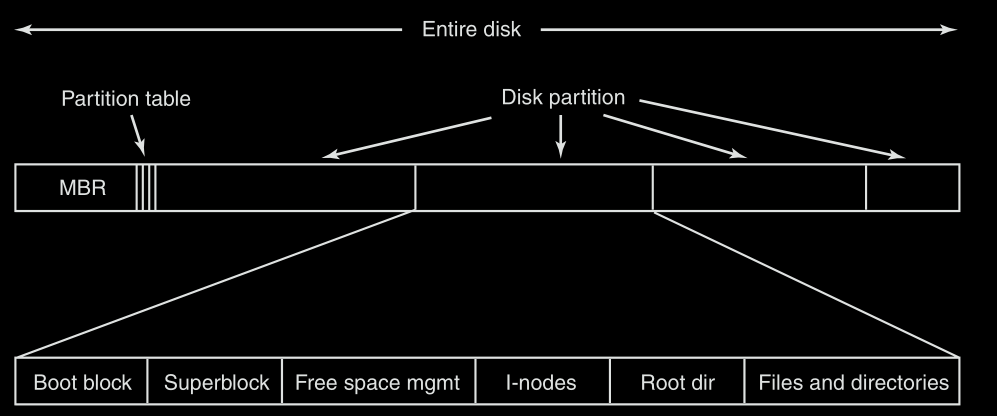


Figure Example File System Layout

* Superblock: Contains key parameters about the file system. Is read on boot

**File Implementation**:

* Contiguous allocation: Allocating file space in one contiguous block
  + Pros:
    - Easy to implement, high performance, seek can read the entire block at once
  + Cons:
    - Disk space becomes fragmented when files are deleted
  + Mainly used in blu-rays and dvds
* Linked-List allocation:
  + Store files as a linked list of disk blocks
    - Poor for random access as you will have to iterate over disk blocks to find a particular file location
  + Most efficient linked-list method is to store points of disk blocks in a File Allocation Table (FAT)
    - File can be put into memory from disk by concatenating disk blocks
    - FAT table is essentially sized to cover the whole disk
  + Does not scale to large number of disk blocks because of the memory requirement for the entire table is proportional to the disk size itself (significant!).
    - Table entries for each disk block are kept in memory… does not ignore closed files for example
* **I-Nodes** (Index node, inode): Uses index node data structure to keep track of what blocks of memory belong to a file
  + I-nodes occupy memory only when the corresponding file is open
    - Typically far less overhead memory cost than FAT
  + I-nodes only have room for a fixed number of addresses? How can we fix this?
    - Point last disk address to the next block of addresses
    - Two+ blocks containing disk addresses
    - Bitmaps for free blocks of memory & free i-nodes on disk

**Directory Implementation**: They are stored like regular files, but are given special meanings by the file system

* List of directory entries
* Each entry consists of:
  + File name, attributes, file i-node number
  + Maps human-oriented file name to system-oriented name
* Fixed vs variable size entries:
  + Fixed: Too small or wastes space
  + Variable: Can cause external memory fragmentation… but we can compact this when the blocks are in RAM
* Searching Directories:
  + Linear scan
    - + Cache to speed up search!
  + Hash lookup
  + B-tree 🡪 For 100’s of thousands of entries
* Storing file attributes:
  + Entries simply refer to i-nodes which hold file attributes (UNIX)
  + Disk addresses and attributes in the directory entry (FAT)

FS Block Size:

* Small size: Wastes less disk space due to less internal fragmentation
* Larger size: Less FS metadata, easier random access

Shared Files: The same file needs to be accessed by different users on a system, thus the file system structure has to become a directed acyclic graph

**Journaling**: A log of operations is kept so that even if an OS crashes, the file system operations can still be completed

* Operations such as remove may be dependent on the order of operations in a crash.

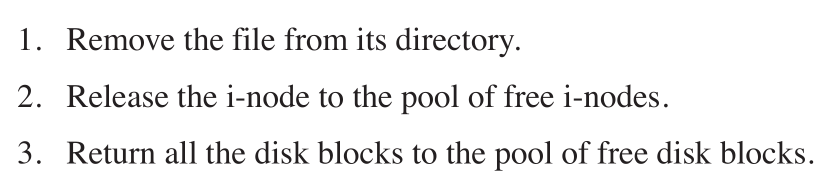


Figure Remove file operation. if the computer does not crash, order does not matter. However you may have a memory leak if it crashes between these operations

* File systems will log all pending actions in sequence before starting them.
  + Journaled operations must be idempotent
* File systems can also designate certain sequences as atomic transactions

**Virtual File System (VFS):** We want to support multiple types of files on our file system

* Framework that separates file system independent and file system dependent code
* Different types of file systems can be mounted on the virtual file system
  + VFS can transfer control to mounter file system that operates on the specific storage structure
  + Indirection layer between syscalls and code for specific file system types that implement the syscalls
* Vnodes represent the underlying i-node in the mounted file system

VFS and Vnode structures:

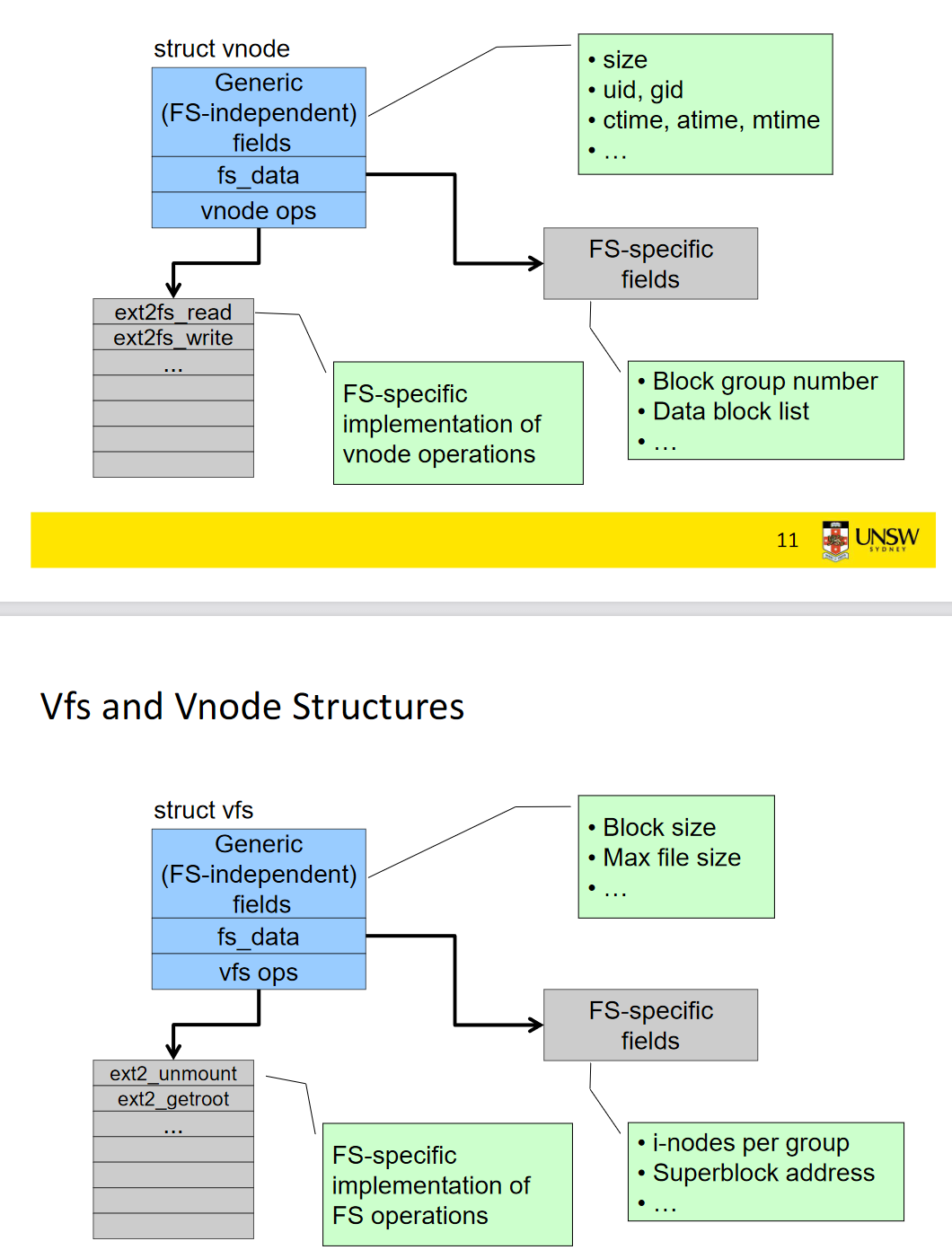


Figure Vnode and VFS structures

* FS-specific fields deal with partitioning for the VFS to point to the right FS

File Descriptor: Specifies which file to operate on

* Associated States:
  + File Pointer: Fp determines where in the file the next read or write is performed
  + File Mode: Opened, read\_only, etc..

There is an interface between the application’s read(fd, data) and write(fd, data) function calls 🡪and the VFS implementation that calls particular file systems

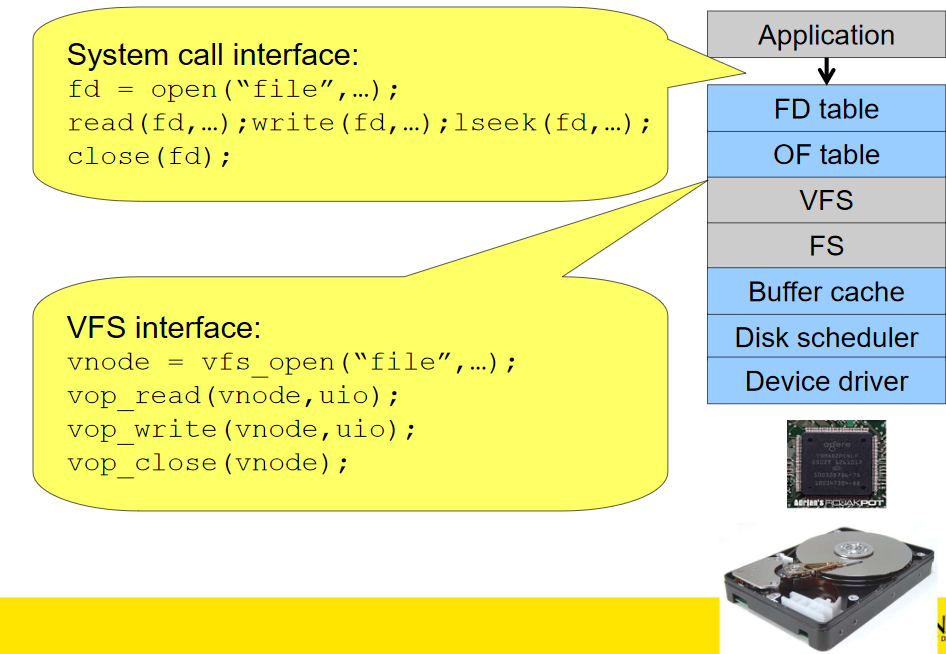
* File descriptor and Open File (OF) table separate application (syscall) and VFS
* Uio variable in VFS interface gives file pointer offset (location in file)
* 

Figure Layers of Abstraction for file systems

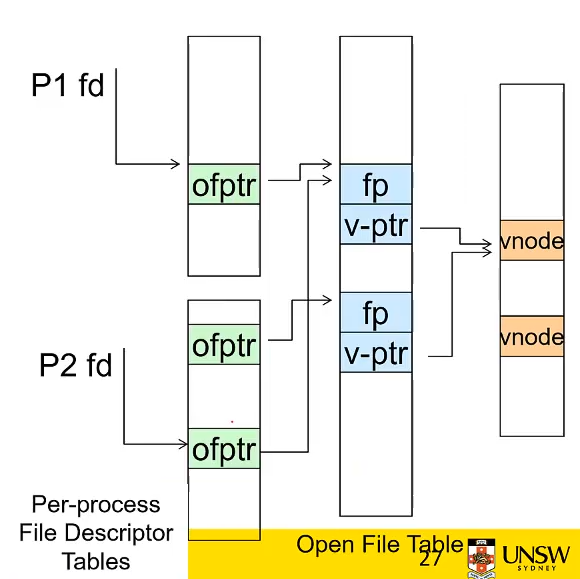
Vnode File Descriptors:

* We can’t use integers for file descriptors because different processes may have different ideas of what FD=1 means (stdout: console or file?!)
* Open File arrays for each process? It will contain fp, v-ptr, etc
  + Fork and Dup2 cause child threads to share open files from the parents!

File Pointer: Created when a file is opened by a process. Logs the offset from the start

**Buffers**: Temporary storage for transferring data

* Producer / Consumer: Mismatch in bandwidth / rates of data processing
* Incompatible units of transfer / size of update: E.G. Disks use blocks, but applications write bytes/bits.
  + Blocks are loaded into kernel RAM buffer. We write to that buffer, the entire block is saved back to disk
* Knowledge of the underlying FS can optimize read/write calls
  + E.G. Sequentially writing # bits == 1 block in a write syscall, means we can write to the whole ‘block’ in the buffer, and easily insert the buffered block into the disk
  + Can implement read-ahead loading in buffer for sequential access



* The files descriptors all point to the same file (vnode). Operations manipulate the same file
  + Differences:
    - File descriptor used to manipulate
    - File pointers can be shared with another process or independent

**Caching**: Fast storage to temporarily hold data to speedup repeated access to the data

* Main memory can cache disk blocks
* Merged into buffer-cache

Buffer Cache:

* UNIX: Hash device# & block#, check if match in buffer cache 🡪 Operate on buffer cache data
  + If hash collision, load block from disk into buffer cache (replacement)
* How to choose buffer cache entry to replace?
  + First-In-First-Out?
  + Least Recently Used (LRU)🡪 timestamps, but how strict do we want to be?
* OS will bias LRU replacement such that critical data is not kept in the buffer cache forever
  + Minimizes possible loss from corruption, power failure, etc.
  + Prioritizes metadata compared to data blocks (corruption of file system is much more devastating than corruption of a single file)
    - UNIX flushd flushes all modified blocks to disk approximately every 30s
    - ^ Called the write-back approach
* Write-through cache: Modifications to buffer are immediately written to the disk
  + Reduces opportunity for corruption between write and saving on disk, but much slower
  + Used in USB

Memory Management: OS manages part of the memory hierarchy (different types of memory such as RAM, disk memory, processor registers, etc.)

* Programs need their own virtualized view of memory

Memory Abstractions:

* No abstraction for memory
  + Works on microprocessors that only run one program
  + Requires you to know what variables are addresses.
  + Makes loading two or more programs into memory rather difficult as well.
  + Never trust the user: The user can screw over the OS with this system
* Address Spaces: A set of addresses that processes can use to address memory
  + Base and Limit Registers: Disused in favour of more advanced schemes in modern CPUs
    - Two hardware registers: Base and limit
    - Base: Physical address where the program begins in memory
    - Limit: Maximum address of the program
  + There are potentially too many concurrent processes running on modern end-systems to keep everything in memory at once; this is termed memory overload
* Approaches to memory overload: Swapping and Virtual Memory
* Swapping: When a process is added, it’s added to the lowest available address in memory that can contain the entire process.
  + Holes can be processed through memory compaction; all the memory addresses are moved downwards as far as possible.
    - Compaction takes a lot of CPU time
  + What if you try to grow and there is no available space on top of the current process on the stack because another process is occupying it in memory?
    - Process will have to moved, a hole must be created, or the process must be killed
    - Usually, a little bit of extra space is allocated to a memory block. As it is assumed that processes will grow by dynamically allocating memory
  + Memory Management, how do we know a block of memory is free?
    - Bitmaps: A bit maps to a block of memory. 0 if free, 1 if occupied
    - Linked Lists: Linked list of allocated and free memory segmets. Algorithms for finding suitable holes in memory are:
      * First fit: scans until the first fit is found
      * Next fit: Same as first fit, except it keeps track of where if finds a hole
      * Best fit: Searches list end-to-end for the smallest hole that is adequate
      * Quick fit: separate lists for holes of common sizes
* **Virtual Memory**: Each program’s address space is broken into chunks called **pages.** 
  + Page: A continuous range of addresses, mapped onto physical memory
    - Pages do not have to be in physical memory, they are loaded in when needed
  + Generalization of the base and limit registers
  + A program has virtual addresses. When a program accesses a memory address, it sends the virtual address to the memory management unit (MMU), which maps virtual addresses to physical addresses through a **page table**
    - Whether a page Is present in memory is denoted by a present/absent bit in the page table entry
  + Allows multiple pages from different processes to map to the same physical memory block
    - Page number: high bits represent the number of the page, whilst lower bits can be the offset in the page

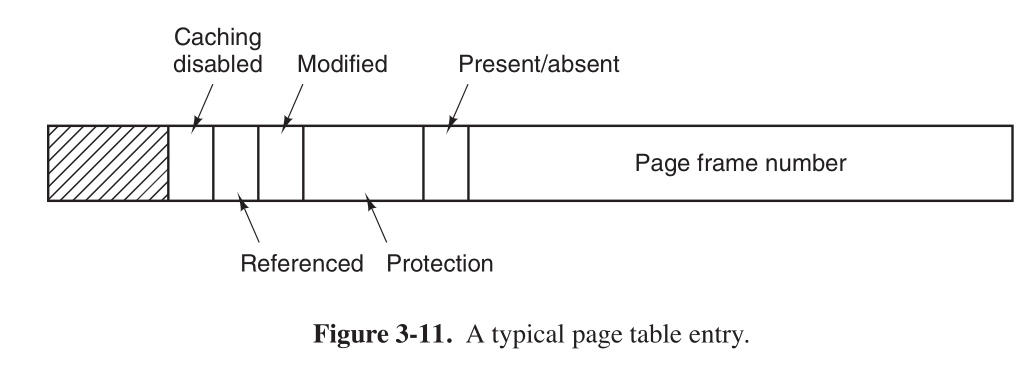


Figure Page Table Entry

TLB: Translation Lookaside Buffer

* Memory cache used to reduce the time taken to access a user memory location
  + Fixed number of slots containing page-table entries
  + Page table queries require to find the frame number, then to check of offset. TLBs reduce the time taken to access memory by only requiring lookup at one location
  + Translate application local page references to physical frame references
* TLB Entry:
  + Contains information about a single page
    - Virtual page number
    - ‘Dirty’ bit (modification bit for pages)
    - Protection code (RWX perms)
    - Physical page frame where the page is located
  + Has one-to-one correspondence to fields in the page table, except from the virtual page number (unnecessary in the page table) 🡪 TLB provides fast access to virtual pages in the page table

ELF file: Executable and Linkable Format

* defines the structure for binaries, libraries, and core files
* formal specification allows the operating system to interpreter its underlying machine instructions correctly
* ELF structure: <https://linux-audit.com/elf-binaries-on-linux-understanding-and-analysis/>
  + Header
  + File Data
* Determines range of virtual address space for a program 🡪 Regions determined by ELF file

Page Table: Basic schemes include

* Use data structures that adapt to sparsity
* Use data structures that only represent resident pages
* Use VM techniques for page tables

Two-Level Page table: Split page number into high / low order bits, use the split bits to index in 2 levels of the page table

* E.G. 32 bits 🡪 10 bits PT1 , 10 bits PT2, 12 bits offset

Preemption: Taking away a resource from a process before it’s done with it

Shared Pages & Copy on Write:

Demand paging: Virtual memory management technique

* OS copies disk page into physical memory only if an attempt is made to access it
* Page fault if not in memory 🡪 See lazy loading

Graphical user interface, text

Description automatically generated

Figure General method for demand paging

**Scheduler**: Comes up with a principled way to determine which process runs next

**Multiprocessor Systems:** Multiple CPUs provide better performance for approximately the same cost by sharing the other expensive hardware. (memory, storage, etc)

* Assumes parallelizable workload to be effective
* Objectives of multiprocessor systems:
  + Enables parallel processing
  + Fault tolerance
  + Boost execution speed
  + Application matching
* Refers to the hardware architecture that allows multiprocessing

Shared memory multiprocessors: More than one processor sharing the same memory

* Assumes workload can be parallelized.
* Assumes workload is not I/O-bound or memory-bound.

Amdahl’s Law: Adding CPUs in the parallelizable proportion of code speeds up that section by N times for N CPUs

* Given a proportion P of a program that can be made parallel, and the remaining serial portion (1-P). Speedup for N CPUs
* Allows us to compute the maximum possible speedup using the limit as N->infinity

Minimizing the serial sections allows for the greatest performance increase

**Types of multiprocessor:**

* Uniform Memory Access (UMA): Access to all memory occurs at the same speed for all processors
* Non-uniform memory access (NUMA):

Mainly using UMA for OS3891

**Uniform Memory Access** (UMA):

* Bus-based: Multiprocessor on a single bus connected to memory
  + Bandwidth of the bus becomes an issue. Not infinitely scalable by adding more CPUs because you reach the maximum threshold of the bus
  + Can be improved by adding cache memory (if enough accesses use the cache instead of the bus)
  + Alternative bus-based architectures do exist
    - Can improve bandwidth availability
    - Doesn’t eliminate constraint of limited bandwidth
* Cache Consistency: if we have a cache per CPU, then how is behavior defined when one processor attempts to fetch data from memory that was supposed to be written by another CPU (instead, it’s in the other CPU’s cache)
  + No consistent view of memory?
  + Hardware needs to ensure the cache is consistent
    - Cache bookkeepinging ensures that memory is copied from the cache/memory with the most recent access. Get consistent view
    - This validation to get copies still cause transactions on the bus.

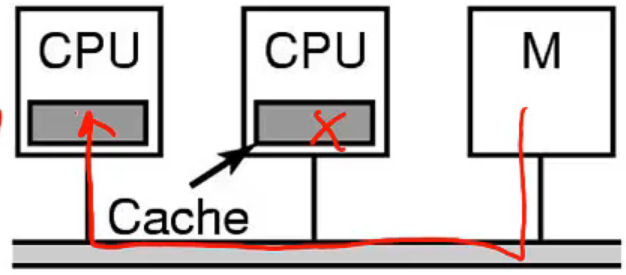


Figure Data saved on CPU cache and not main memory. How do we know to fetch from the other CPU’s cache?

* Modern multi-core processors use processor hierarchies with corresponding caches

For distributed systems: Don’t assume:

* Reliable and secure networks
* Zero latency
* Infinite bandwidth
* Network topology is unchanging
* One administrator
* Zero transport cost (e.g. bus transactions w/ caching)
* Everything is homogeneous

**OS configurations for multiprocessor systems**:

* Individual OS for each processor
  + Historically simple to implement using a uniprocessor OS
  + Avoids CPU-based concurrency by not sharing
  + Scales due to no serial sections
  + Modern analogy: Virtualization in the cloud
  + Memory needs to be partitioned between operating systems
* Symmetric multiprocessors: OS runs on all processors
  + Issue: Real concurrency in the kernel, requires careful application of synch primitives
  + Possible solutions?
    - Lock for OS access. The lock becomes a bottleneck though
    - Identify large, independent parts of the kernel and make each of them their own critical section
      * Deadlock conditions if lock ordering schemes for kernel regions are not correct
      * Lock hierarchy 🡪 Creates bottlenecks
      * Need to divide into more and more complex lock (OS) structures based on optimal lock hierarchy
  + Lock contention can seriously bottleneck system performance
    - Especially with the single-threaded serial OS code
* Writing complex, large, parallelizable code is really challenging..
  + Linux 1996 (big lock) 🡪 2011 big lock removed
    - Multi-step evolution with more subsections of the kernel being migrated out of the big lock
* Symmetry in multiprocessor systems:
  + Can be symmetric or asymmetric
  + Non-uniform memory access
    - Memory access depends on the memory location relative to the processor
    - Benefits are generally limited to workloads like servers where there is a strong association between data and user
  + Clustered multiprocessing (compute clusters)
    - Processors are combined into fast local networks with nodes (server computers) running OS instances

Possible Multiprocessor Design Paradigms:

* Master / Slave : Master CPU assigns tasks to slave CPUs
* Tightly coupled systems: Multiple CPUs are connected at the bus level
  + Can have a memory hierarchy of local and shared memory
  + Multi-core computing (multiple processors on a single chip) is a form of this
  + Easy load balancing / distribution
  + Bottlenecks from sharing common resources on one or more buses
* Loosely coupled systems: Multiple standalone single or dual processor commodity computers connected via a high speed communication system (e.g. gigabit ethernet)
  + High performance for each individual processor but does not enable for easy real-time balancing of the load among processors
* Multiprocessors with global data manipulation: Has common resources and local resources (e.g. Caching)
  + Partition local memory into global and local data
    - Global data: modifiable information accessible by all processors. Duplicated across processors. E.g. Lock in cache
    - Modifications to global data in local memory causes a hardware broadcast to all other CPUs along the system bus
    - Local data can be exchanged by message-passing

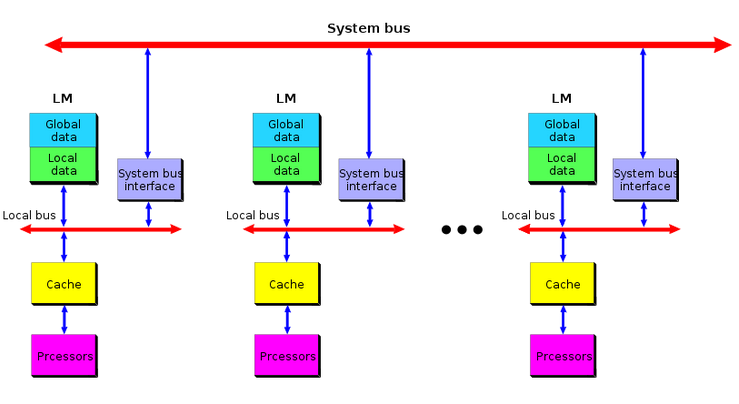


Figure Multiprocessor system with global data multiplication

Synchronization primitives for multiprocessor systems:

* Disabling interrupts does not work (does not protect critical sections from other CPUs)
* Mutex test and set? 🡪 this is a spinlock thus blocks the bus for all other CPUs
  + Atomic within the CPU. Does not work without hardware support because otherwise two processors can read the lock as available before the set instruction executes
  + Hardware needs to prevent the interleaving of the bus transactions to memory by individual CPUs in the machine
    - Block CPUs from accessing the bus during a TSL? Results in busy wait by all other CPUs though (bad!) = causes bus contention 🡪 slows down all CPUs
* Test and set on cached copy of memory in CPU
  + Hardware ensures that only one copy of memory location undergoing a TSL can exist in a single CPU
  + Other copies are invalidated during TSL
    - Most recent version gets transferred to TSL’ing CPU
    - This causes bus traffic but does not block
  + TSL 🡪 Main copy in test and set happens atomically, locally in the cache
  + How do we modify the TSL instruction to not perform TSL every time the lock is unavailable:
    - See explanation Figure 23 (TSL with SMP)

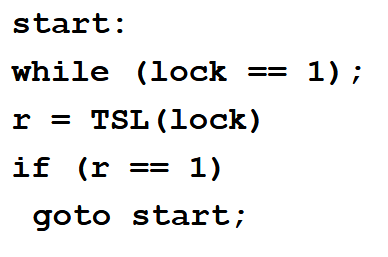


Figure TSL with SMP

* Hardware needs to ensure mutual exclusion when a lock is set for a particular CPU

TSL Cache Sequence: Read before TSL

1. All waiting CPUs wait until the CPU with the lock releases it
2. Cache handler updates all CPUs with lock == 0
3. One CPU executes TSL first (ensured by hardware).
4. Cache handler updates lock status to 1
5. All other CPUs that did not execute TSL see r == 1 thus they repeat the loop

* Note: The above instruction uses the bus for updates on the lock status ( r ), but the CPUs are not waiting on the bus. Therefore performance is better with this model
  + CPUs are waiting in the cache instead
* No race conditions (acquisition with TSL)
* Critical section performance degenerates with more CPUS @ step 3
  + All the CPUs execute TSL and therefore compete with the CPU that’s holding the lock
  + TSL performs poorly once there are enough CPUs to cause lock contention
* Allows lock to be shared read-only in all caches until its released

Spinlock vs Blocking: Uniprocessor case

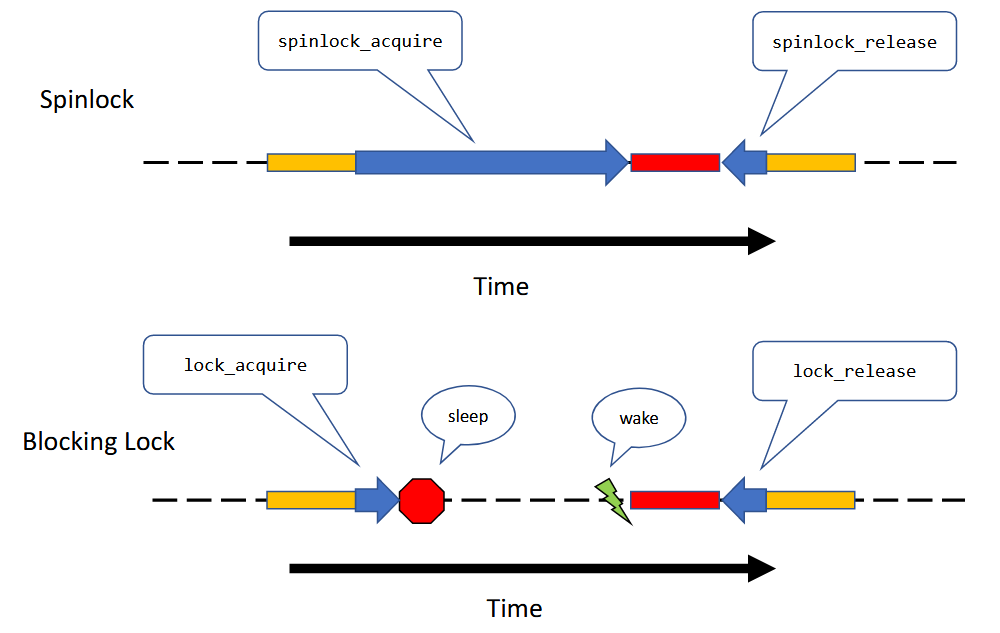


Figure Example Spinlock vs Blocking Lock (wakeup through interrupt): **Uniprocessor case**

* Time is saved by blocking as the thread does not use CPU time by spinning
  + Especially when the scheduler chooses the spinning thread whilst another thread has the lock (and is in the critical section)
* Doesn’t make sense to spin on a uniprocessor
* For SMP systems:
  + Makes sense to spin if spinning time < thread wakeup from block time on another CPU E.G. Critical section is small
    - Blocking and switching takes time (save context and restore another)
    - Switching wastes CPU time directly
    - Spinlocks are not effective with longer wait times on the critical section (e.g. waiting on I/O)
  + Tradeoff:
    - Lock needs to be held for less time than switching overhead for spinlocks to be efficient
  + TLDR:
    - Short critical section 🡪 Makes sense to spin since it will be faster than blocking wakeup
    - Longer critical section 🡪 Use blocking locks to do computations on other threads whilst blocked
* Preemption and spinlocks: Bad for multiprocessor systems because all the other CPUs will have to wait for the preempted thread that holds the lock to release the lock
  + Effectively makes the critical section as long as it takes for the lock holder to be scheduled again
  + Arbitrarily long critical section
  + Therefore spinlocks need to disable interrupts for the lock holder to disable preemption

**Scheduling**: Deciding what to run next

* Has a dramatic effect on the perceived performance of the system (and also the actual performance).
* CPU-Bound process:
  + Spends most of it’s time computing
  + Time to completion largely determined by received CPU time
* I/O bound process:
  + Spends most of it’s time waiting on I/O
  + Time to completion largely determined by I/O request time
* We can maximize performance by switching between CPU and I/O bound threads
  + Difficulty arises from the fact that the CPU goes through multiple phases of execution between CPU and I/O bound
* If you choose to schedule an I/O bound process, it will delay the CPU bound process by very little because most of the time is spent waiting
  + Compare that to if you delay the I/O bound process by scheduling a CPU bound process. You delay the time it takes for the I/O bound request to send the request
  + Favoritism: I/O bound processes > CPU bound processes
    - I/O bound processes affect reactivity for the user
    - System looks responsive

Preemptive vs non-preemptive scheduling:

* Non-Preemptive: Scheduler cannot preempt CPU time from the current running thread unless the thread voluntarily yields to the CPU
  + Used in simple systems where you don’t want the timer tick to context switch
  + E.G. washing machine, simple embedded devices
* General programming systems implement preemptive scheduling where the OS can force the CPU to execute different threads
* No-algorithm fits all computational environments: many tradeoffs
  + Batch systems: No users directly waiting… (not that common xD)
    - Optimize for machine performance. Can use non-preemptive
  + Interactive systems: users directly waiting
    - Optimize for interactivity. For perceived performance
  + Realtime systems: Must schedule jobs such that all jobs (mostly) meet their deadlines
    - E.G. Computer for airbags in car
* What are our goals?
  + Fairness: Define fairness goals
  + Policy enforcement: Schedulers should ensure that the fairness policy is carried out
  + Balance/Efficiency: E.G. I/O bound processes are prioritized over CPU bound processes
* Goals are based on what kind of system we are dealing with: interactive, batch or real time

*Types of schedulers:*

* Round-Robin Scheduler: Assigns a time-slice to each process
  + Implemented with a ready queue and a regular timer interrupt
  + Easy to implement and used when basic preemptable scheduling is needed
  + Choosing a smaller time slice 🡪 Gives the illusion of more responsiveness but the overhead for switching becomes comparatively larger
  + No idea of priority
* Priorities: Assign priority to processes and run the highest priority first in the scheduler
  + Higher priority thread should always be chosen in lieu of lower priority thread
  + Implementation using multiple priority queues, using round-robin on each queue
    - Tweak priority based on age
  + Con: low priority can starve

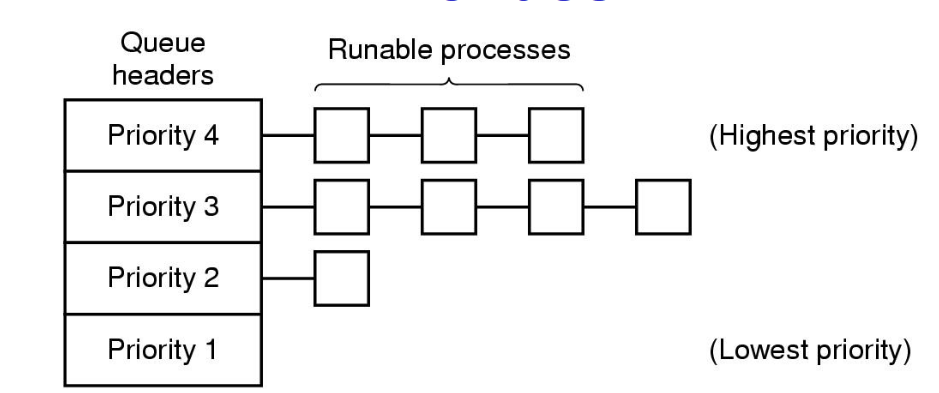


Figure Mutliple priority queues

* Unix Scheduler: Priority based scheduler
  + Recalculates priorities every second and penalizes CPU bound threads
  + Priority = CPU\_usage + nice + base
    - CPU\_usage: number of clock ticks; decays over time to avoid starvation of CPU bound to I/O bound threads
    - Nice: Value given to permanently boost or reduce priority. Set by user
    - Base: Hardwired priority values for I/O bound system activities

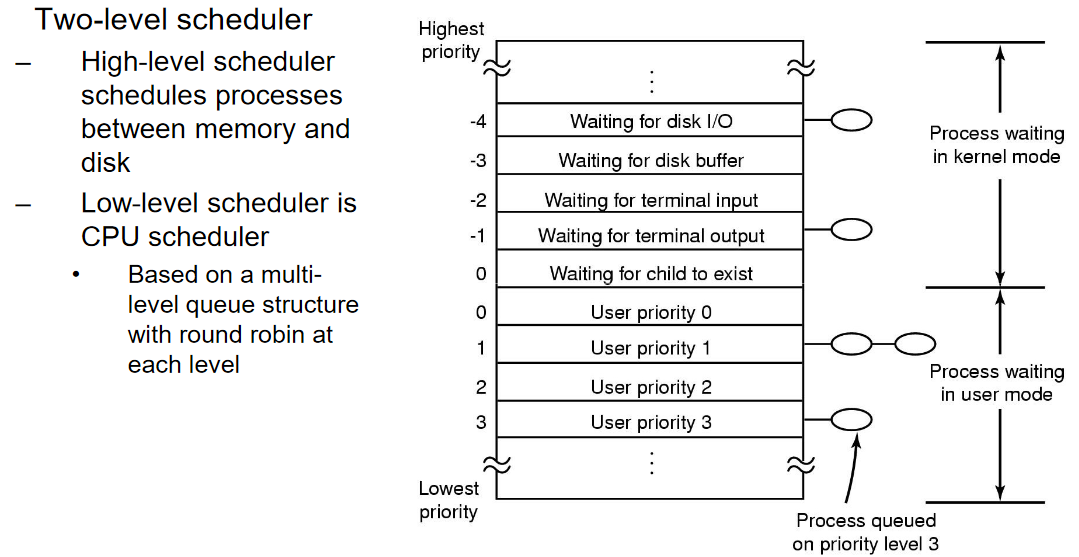


Figure UNIX Scheduler example. Is iterated on every couple of years :P

Multiprocessor Scheduling:

* How is it different from uniprocessor scheduling?
  + Each processor needs to be kept busy
  + Consider this: given X processes and Y CPUs, how does each CPU choose which thread to run on itself
* Single shared ready queue:
  + CPUs run the highest priority thread in the queue when they go idle
  + PROS:
    - Simple 🡪Shared data structure kind of like the uniprocessor priority queue that has been synchronized to avoid race conditions
    - Automatic load balancing: threads are automatically run on the next available processor
  + CONS:
    - Lock contention for ready queue 🡪 This SINGLE shared lock can become the bottleneck
      * Due to frequent scheduling or many CPUs or both
    - Performance is dependent on the related entries in each CPU cache
      * Does not consider optimizing scheduling for CPU caches 🡪 Does not specifically use the cache footprint from previous context switches if the scheduler simply loads onto the next idle CPU
* Affinity based Scheduling: Scheduling that biases decisions such that threads tend to run on the same CPU, maximizing performance gains from caching
  + Create affinity between a thread an a particular CPU

Affinity Scheduling Approaches:

* Multiprocessor scheduling algorithms are still an active field of research!
* Multiple queue SMP scheduling: Per-CPU data structure
  + Each CPU has its own ready queue
  + Reduce lock contention on that queue as usually only the local CPU accesses the local scheduling queues
    - Other CPUs access other queues when they don’t have anything else to run, so they get work from the other CPUs

OS I/O

* There is a lot of variance for all the different possible I/O devices that computers need to use
  + Different properties and interfaces
  + Problem: How can we standardize a uniform and efficient approach to I/O
* OS should be independent to the lowest level software that controls a device
* The required data processing rate for different kinds of I/O can be orders of magnitudes different
  + E.G. Gigabit ethernet vs keyboards

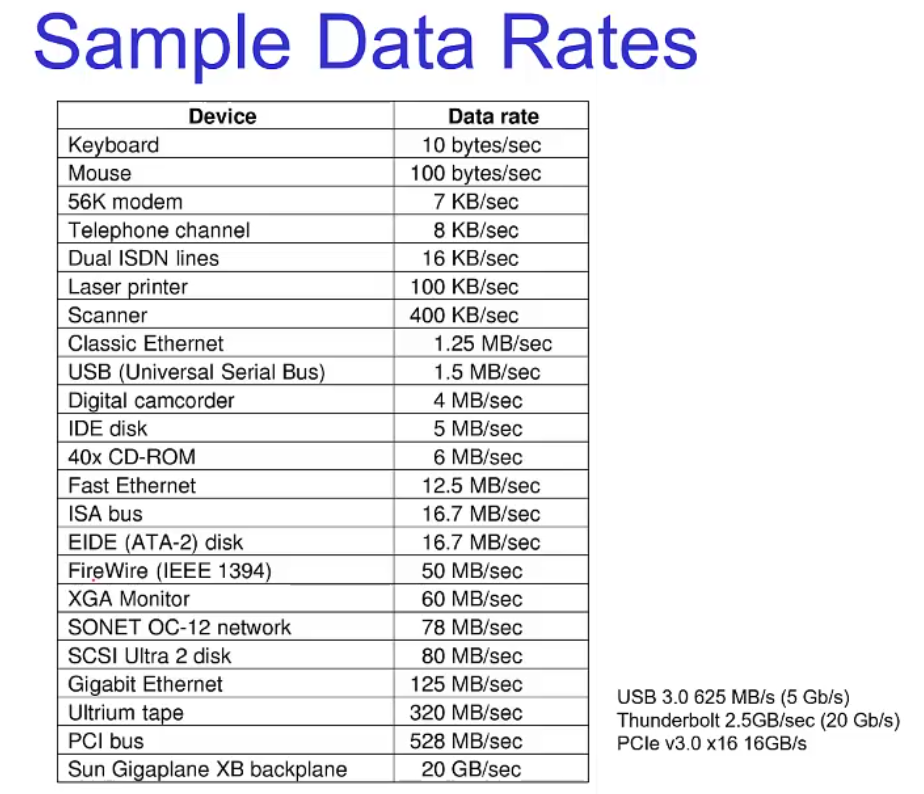


Figure E.G. Data rates for different devices (note: still outdated for recent technologies)

Devices: Managed by device drivers that are usually within the OS

* Device driver: Software components that are part of the OS
  + Drivers are dynamically loaded nowadays (no recompilation needed)
  + Called modules in linux, DLLs (dynamic link libraries) in windows

**Device Drivers**: Translates a device-independent standard interface (open, close, read, write) into appropriate sequence of commands (register manipulations) for the particular device hardware

* Boot-time initialization and shutdown also needed
* Classified into similar categories. Here’s a unix centric view:
  + Block device : I/O read and write entire blocks
  + Character (stream of data) device : Stream of data
* OS defines a standard interface for different classes of devices
* Processing I/O and waiting: Device drivers usually do one of the following depending on the type of device and I/O properties
  + Complete immediately and return to caller
  + Device processes the request and blocks waiting for an I/O complete interrupt
* Divide I/O into device-dependent and independent software:
  + Device Independent software includes: TCP/IP stack, buffer cache management, error reporting, etc.

Driver and Kernel Interface:

* Uniform device interface for kernel code: The kernel uses an interface to interact with different devices
* Uniform kernel interface for kernel code: Drivers use a defined interface for kernel services
  + Device drivers also need to use the interface to access OS functions
  + E.G. Kmalloc, IRQ handler

How does software manipulate I/O Controllers?

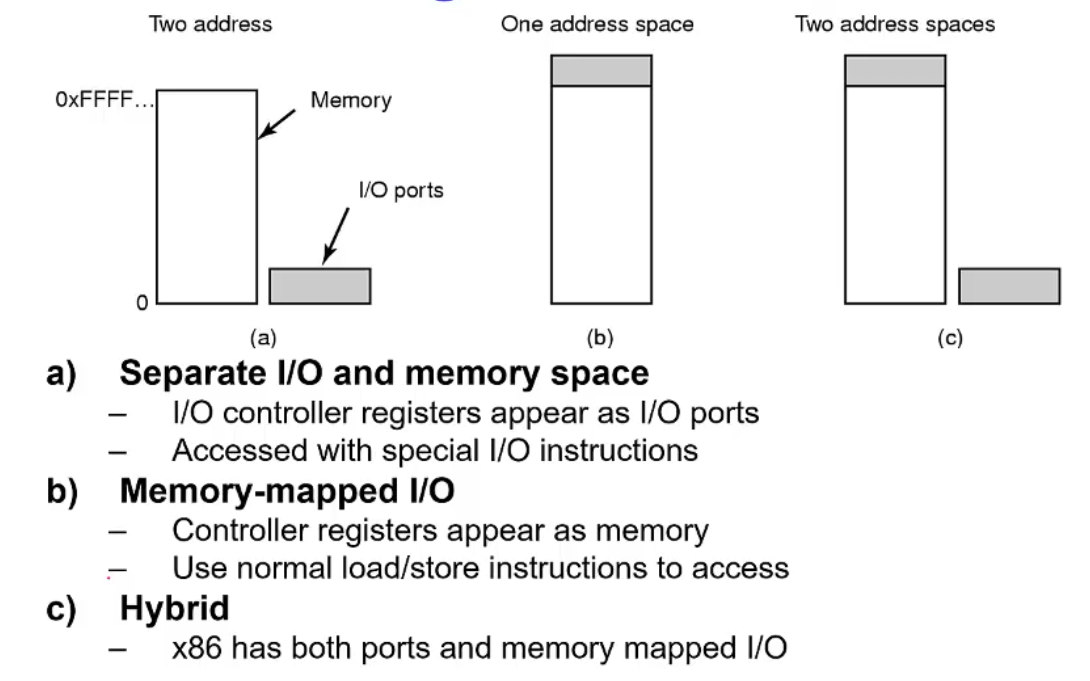


Figure Accessing I/O Controllers: 3 approaches

1. Historical approach: I/O port space and rest of memory are separated
   1. Different instructions for writing to I/O vs memory
2. Modern approach: Physical address space be divided between memory and controllers
3. Hybrid of two above approaches: Used for legacy support
   1. Used in x86

Bus Architectures:

* Single-bus : One bus for CPU, memory and I/O
* Dual-bus: Bus between CPU and memory (to keep cpu busy) and separate I/O bus (much slower rate)
* Buses are isolated as required by hardware for performant parts of the computer

Interrupts:

* Interrupts controllers can be used between devices (I/O) and CPU
  + Controller issues interrupt 🡪 CPU ACK’s interrupt
* Allows CPU to notice when devices need exception
  + E.G. Timer device regularly generates interrupt
* Architecture specific

How does software interact with a device?

* Programmed I/O: Polling / busy waiting on a status bit in the I/O module
  + Wastes CPU time like busy waiting in rest of OS, repeat until I/O is completed
  + Inefficient when the device cannot produce the result immediately
* Interrupt driven I/O: Processor is interrupted when I/O module is ready
  + No busy waiting, but still consumes processor time because you need to handle exceptions for interrupts using vmfaults
  + Too much overhead when working with large streams of I/O
    - E.G. Gigabit ethernet.. interrupt for every byte == RIP processor
* Direct Memory Access: Have an intelligent controller in the device that performs direct memory access
  + Separate from CPU so you don’t waste processor time
  + More efficient as the device can access memory across the bus, instead of faulting memory in and out of the CPU from memory to the device
  + External DMA controllers are generally for older machines as modern devices generally have their own DMA controller

DMA: Direct Memory Access

* Transfers blocks of data to and from memory
* Interrupt is only generated when the task is complete, thus the processor is only invoked at the beginning and end of the instruction.
  + Way more efficient than copying using interrupts; one word at a time
  + TLDR: Reduces number of interrupts 🡪 Less context switches
* Disadvantages:
  + Requires contiguous regions (buffers) for copying
    - Modern networking cards support “scatter-gather” 🡪 Breaks down and joins (formatting) of troublesome packets for DMA buffers
  + Shares memory bus bandwidth with the CPU

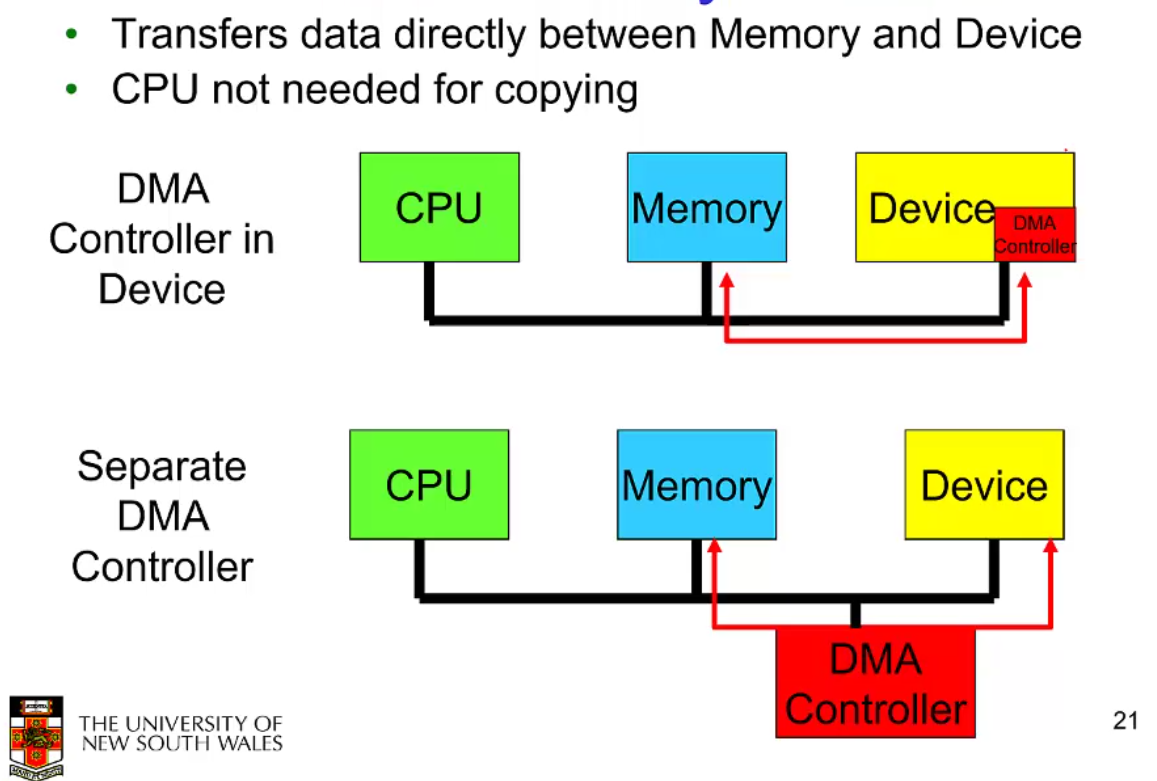


Figure Direct Memory Access DMA Controller

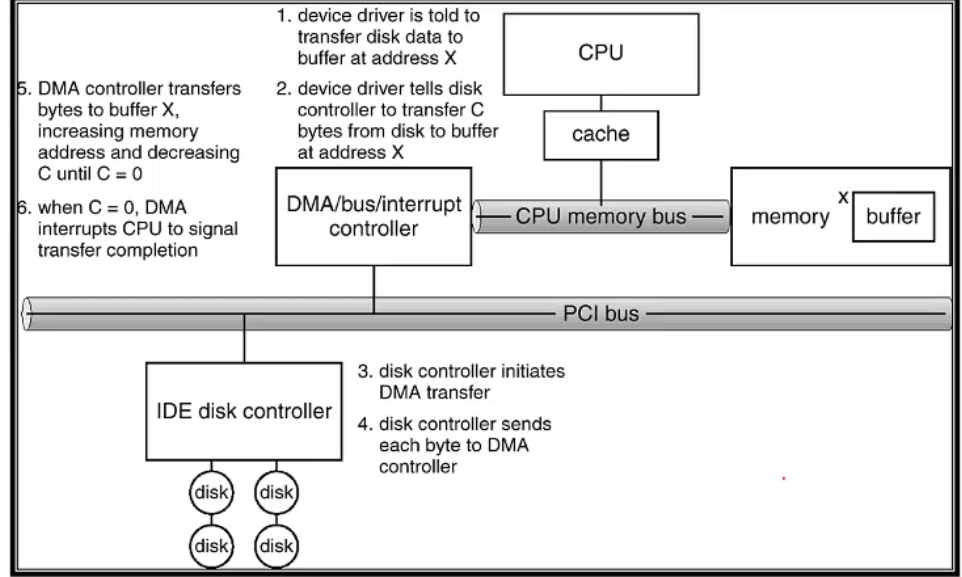


Figure DMA: Hard Disk Controller – Process to perform DMA transfer

OS I/O Design Issues:

* Efficiency is our main priority
  + Most devices are slow compared to main memory and the CPU
  + Can be slightly mitigated by multiprogramming, but often the I/O still can’t keep up
    - Paging can be an issue.. it triggers more I/O operations
* Problem: Diversity of I/O devices 🡪 We wish we could have true generality
  + Different access methods: E.G. disk drives aren’t good for random access
  + Generality often compromises efficiency
* Solution: Hide device I/O in low-level routines and provide abstractions
  + E.G. Make all storage devices 🡪 ‘block’ devices. Thus the file system only has to read and write on block numbers
  + Device drivers present the abstraction.

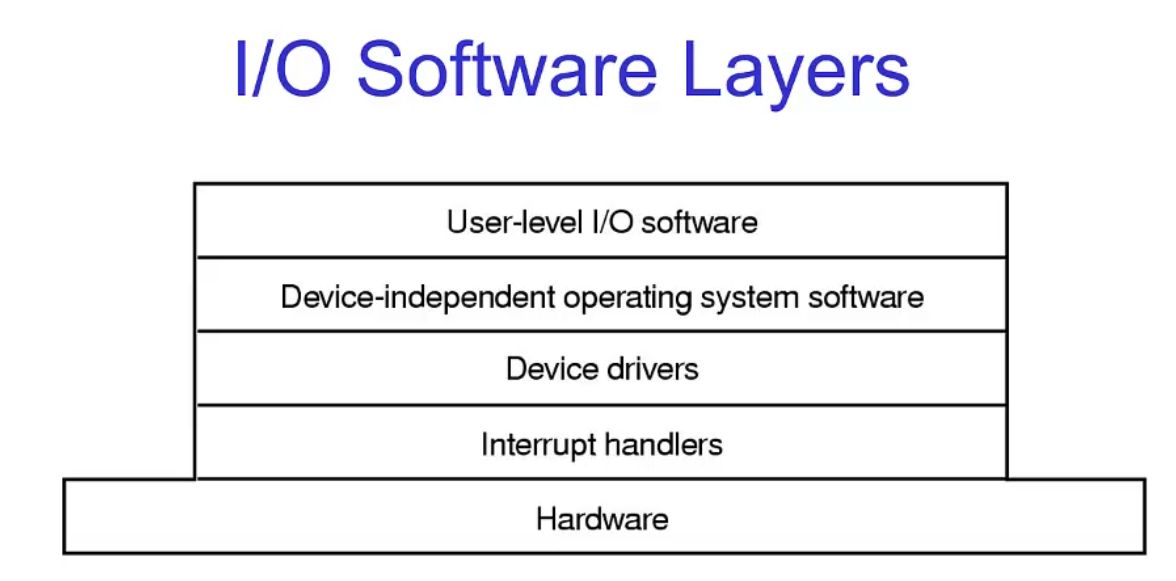


Figure Layers of the I/O Software System

Interrupt Handlers:

1. Save registers
2. Set up stack for interrupt service procedure
3. ACK/Mask interrupt controller, re-enabling other interrupts

Sleeping / Blocking in interrupts:

* In general, interrupt handlers run on the current kernel stack (on top of the current running process)
  + A currently blocked process is the one that generated the interrupt, and is interrupting a currently running (and productive) process
  + Now the interrupt handler is potentially running on the stack of another workload / process
* Possible deadlock for the interrupted process if the interrupt handler blocks
* How do you implement long running code segments in the interrupt handler?

Sleeping / Blocking in interrupts: Solutions

* Top Half / Bottom Half: Uses high priority interrupt thread or divide interrupt handler by turning interrupts back on in bottom half.
  + Top Half: Remains short, acknowledges device.
    - Blocks interrupts
    - Handles hardware operations
    - Adds workload to bottom half (high priority interrupt thread).
    - Then returns and interrupt context is exited
  + Bottom Half: Does deferred processing, runs on kernel stack of currently running process.
    - Cannot block. 🡪 Can be preempted by interrupts, allows for interrupt nesting. The interrupt-off stat is minimized however; additional added work simply adds to the bottom half
  + Main aim: Low interrupt latency. 🡪 Still allows for longer interrupt processing
    - Cons: The bottom half can keep growing if top-half keeps interrupting

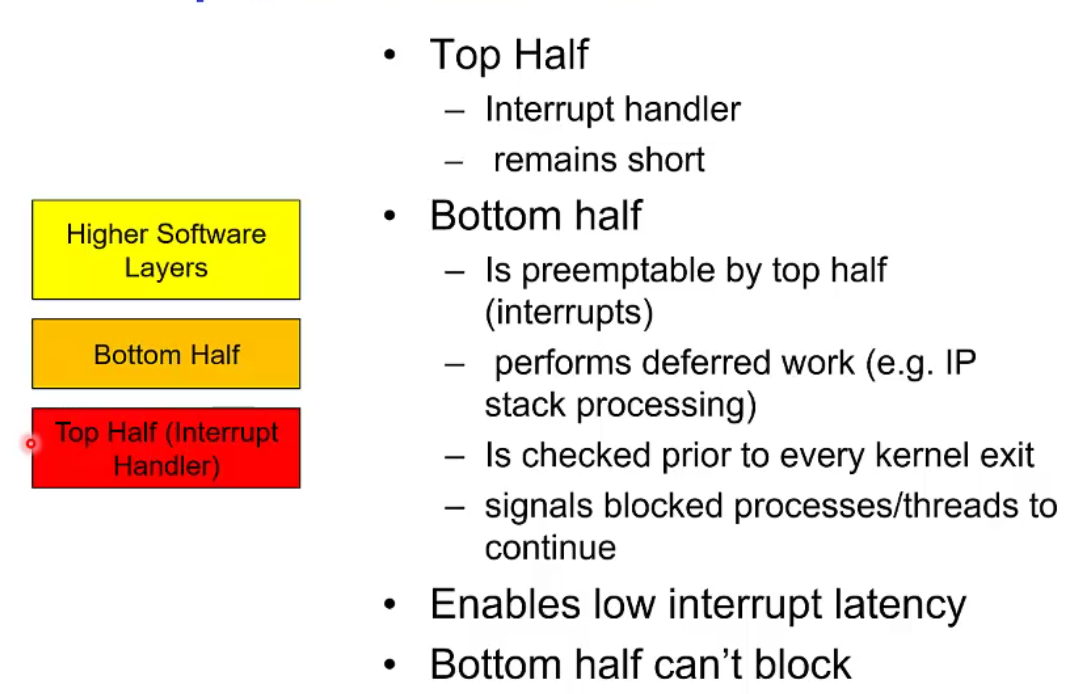


Figure Top Half and Bottom Half

* Deferring work on In-Kernel Threads: non-blocking interrupt work is scheduled to an interrupt thread

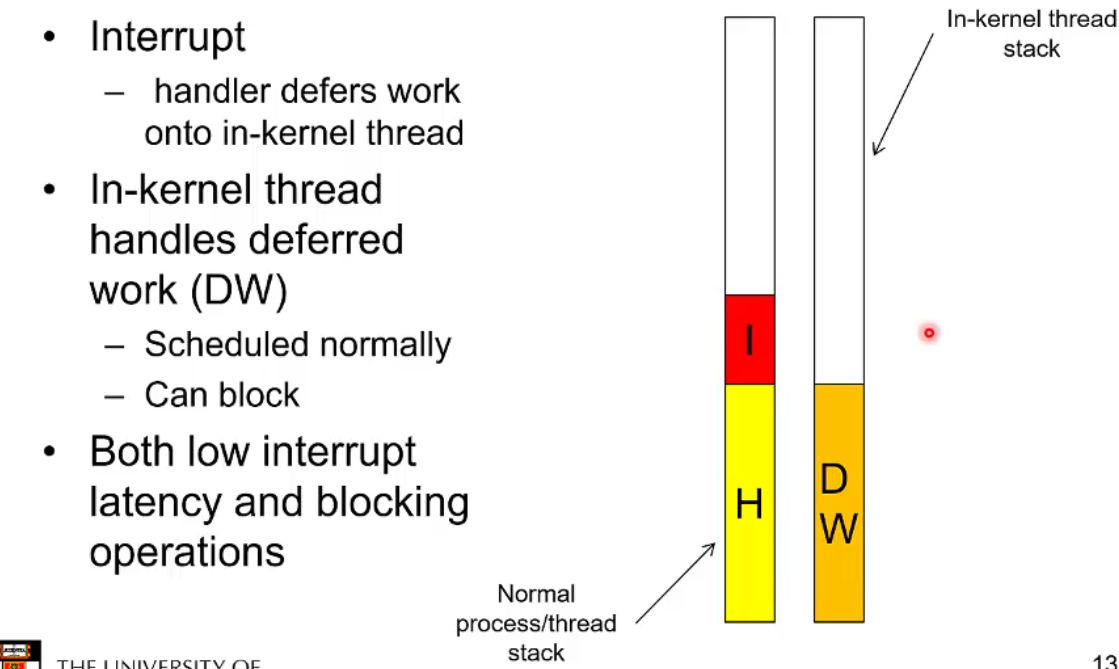


Figure Deferring work on In-kernel threads

* Tasklets and work queues:
  + Used by linux. Allows the programmer to choose between top/bottom half vs deferring to a thread

Buffering:

* No buffering: can be used when data is fetched one byte at a time.
  + Slow and blocking
* User-level buffering: OS delivers to the application, a buffer worth of data
  + Issues:
    - Applications only have a view of virtual memory, buffer could be paged out to disk. Interrupt handlers cannot block so data can be lost
    - Could lock buffer in memory (required by DMA) 🡪 RAM can become a deadlock
    - Is the buffer reusable for writing?
      * Generate signal for buffer reuse? 🡪 Complex and fragile
      * Potential waiting on slow I/O to drain the buffer
* Single buffer in Kernel-Space: Can rely on a buffer in the kernel, independent of what the application is doing
  + OS assigns the buffer in kernel memory for an I/O request
  + Can compute/process data and transfer data (usually the biggest time sinks) in parallel, but you have to add the time it takes to copy from the kbuf 🡪 ubuf (generally pretty negligible)
  + Stream oriented case:
    - Write lines to k-buffer 🡪 u-buffer
  + Block oriented case:
    - Write blocks
  + Can provide speedup given memory-copy cost from kbuffer is small
  + Kernel buffer is full? 🡪 Results in dropped blocks/data/packets
  + Speedup over no buffer is:
    - .
    - T : Transfer time
    - C : Compute / Processing time
    - M : kbuf 🡪ubuf memory copy time
* Double Buffer: Use two system buffers instead of one
  + A process can transfer data to or from any one buffer, whilst the OS empties or fills the other
    - Gives us more leeway by not requiring that we read from the buffer as soon as it fills to avoid packet loss
  + Simultaneous memory copy & data processing
  + Speedup over no buffer is:
    - .
    - Memory copy & compute happens in parallel to the transfer

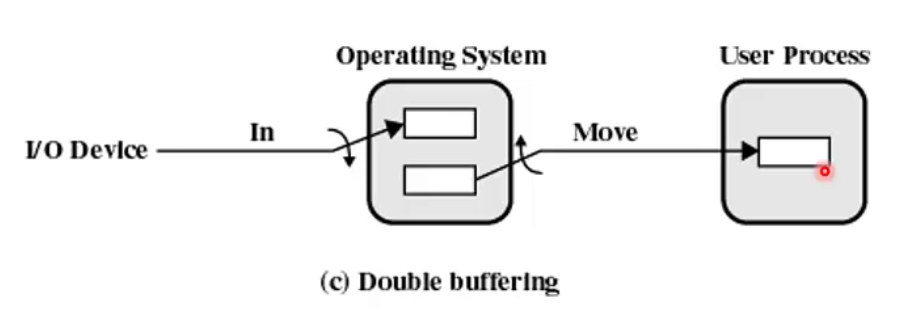


Figure Double Buffer Usage

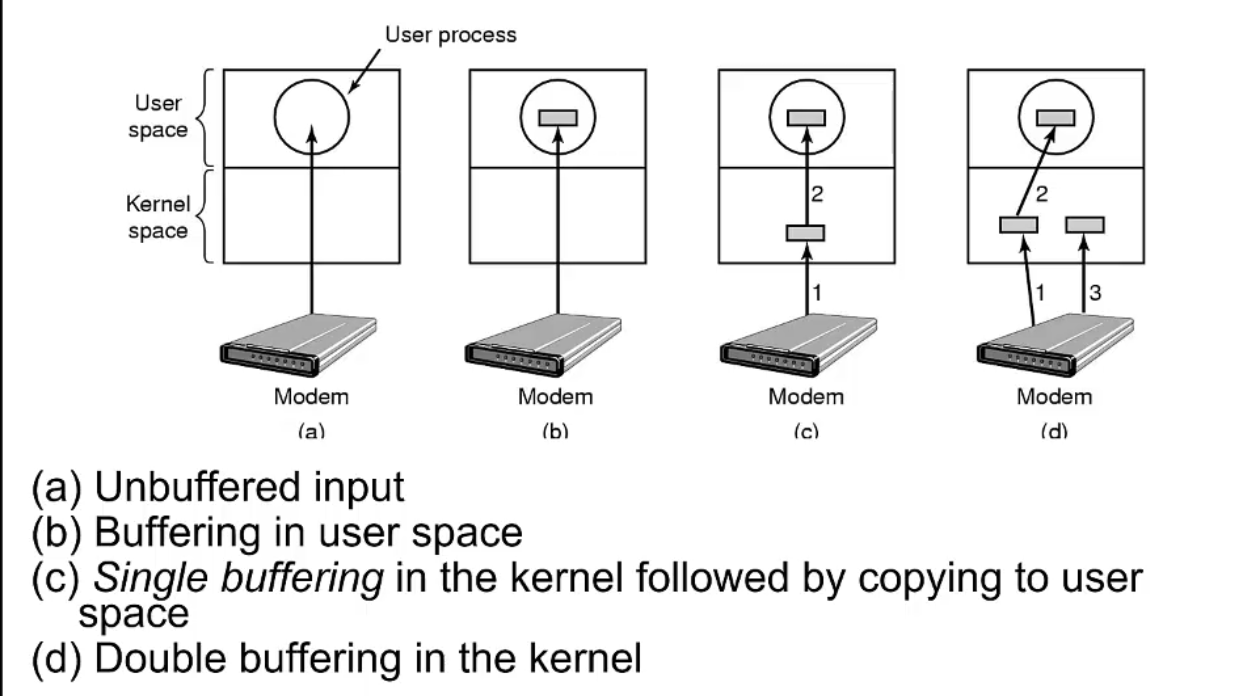


Figure Types of Device-Independent buffering

* Buffering 🡪 Still may be insufficient for bursty traffic such that the application cannot process packets in time with only two buffers
* If two buffers aren’t enough… add more buffers!

Circular Buffering: The bounded buffer producer-consumer problem!!

* Properties of the device are known by the device driver writer… 🡪 Choose the level of buffering required to absorb the burst of traffic

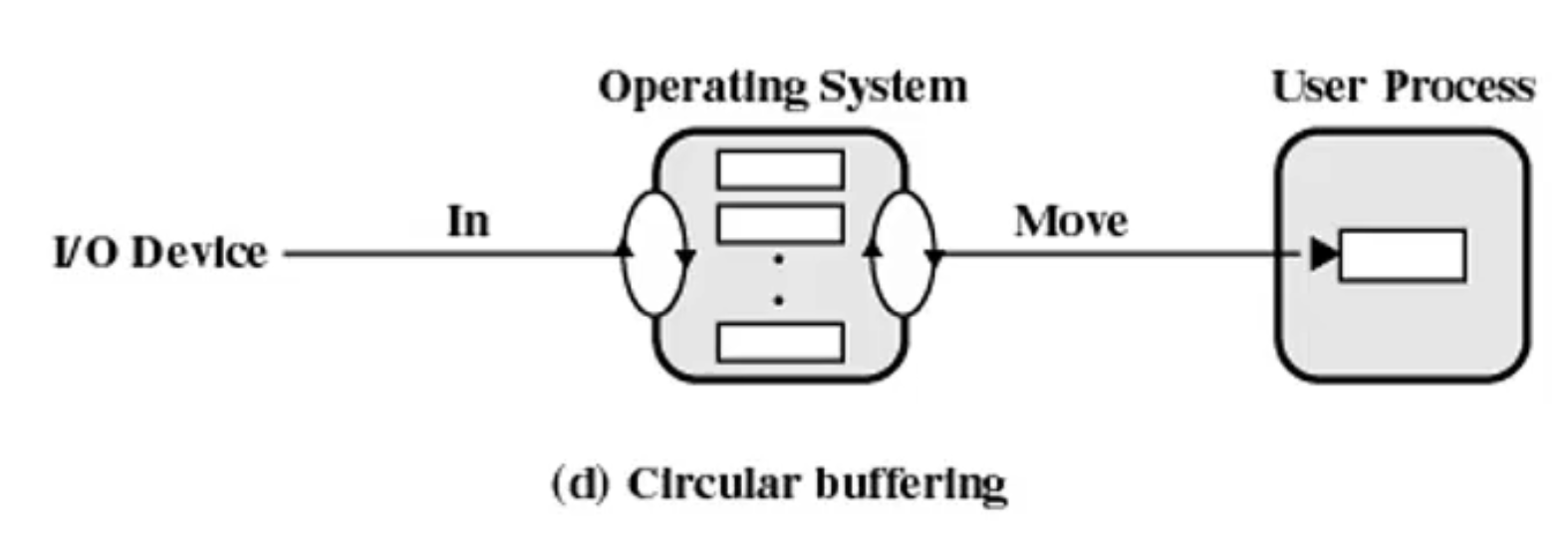


Figure Circular Buffering

Is buffering always performant and efficient?

* Buffering is a performance penalty when the cost of the copy approaches the transfer and compute
  + Seen in fast networking 🡪 Data in network card approaches memory speeds, increasing latency, etc.
  + Copying slows performance

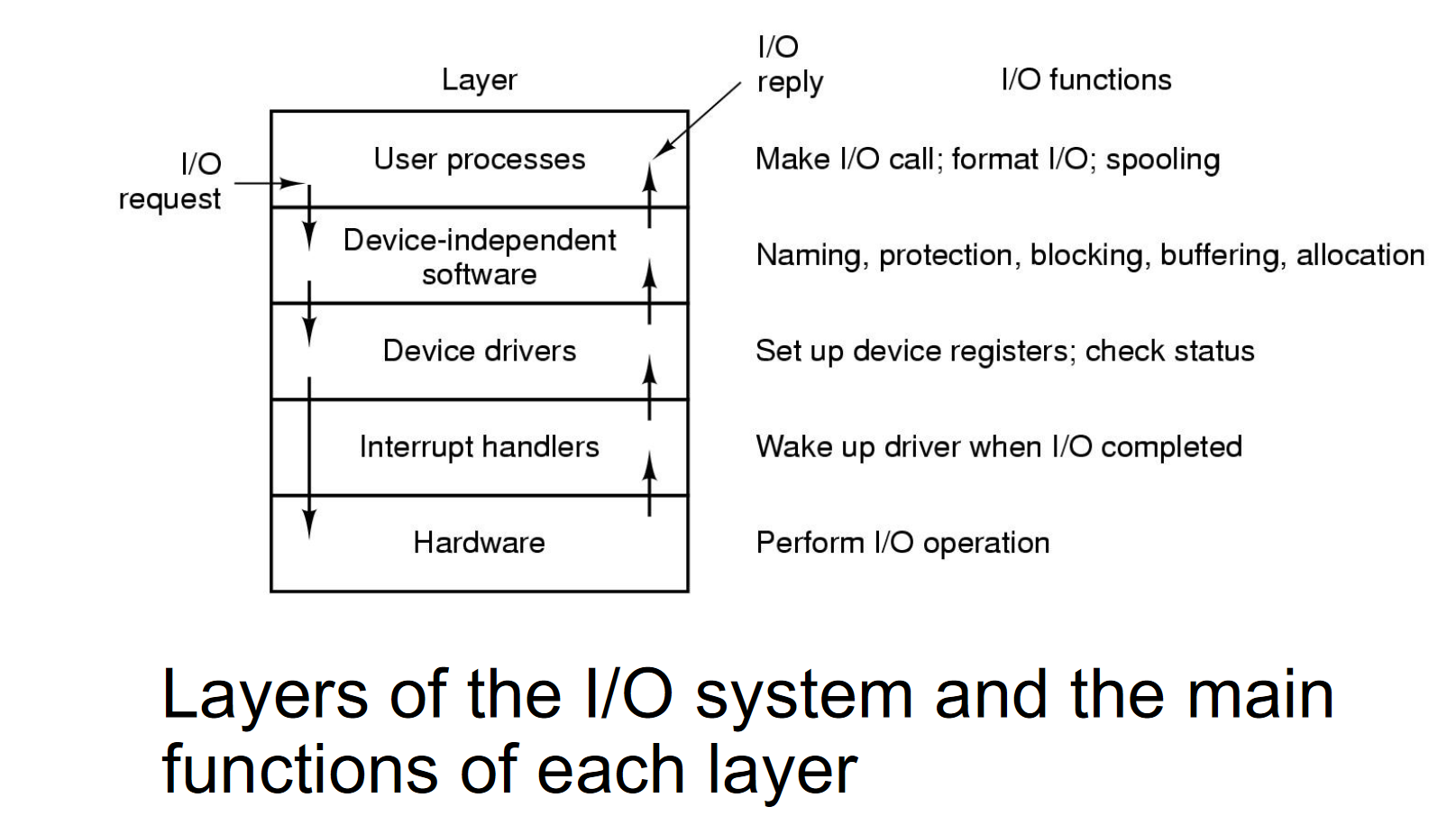


Figure I/O Software Summary

Log Structured FS:

* Historical approach: Write the data into a non-volatile location, store the information in a log, then use the log to write the data to its proper place
  + Used to mitigate data loss due to power outage or other exception
* Log structured approach: Append data to end of log
  + Old version is marked as free space, and the new version is at the end of the log
  + Conceptually, you can find the current state of the FS by replaying the log
  + Log-structured FS periodically writes back to disk based on log checkpoints
* Why?
  + Most writes to the FS are sequential
  + Reads are cheap and writes are expensive: Works for NAND-flash memory as well such as SSDs
    - Log entries are written in large contiguous chunks, after the file system has been cleaned/compacted

Segment cleaning problem:

* Log-based FS have a steady state of spending a significant amount of the disk access time, cleaning up segments
* As CPU and RAM memory becomes larger and more performant, we can increasingly rely on caches instead of disk access

Virtual Machine: Virtualization of a computer system

* Just in time compilation:
  + You have portable code that gives information of the structure of the program
  + JIT keeps a jump-table equivalent, checks if a function is in the table, then loads the memory address of the function into the location in the jump table
* Is the “OS-level” the correct level of abstraction for a virtualized application?
  + Think cloud; does having multiple users share a linux box in the cloud sound like a great idea?

Virtual Machine Monitors: Hypervisor

* Provides scheduling and resource management

Diagram

Description automatically generated

Figure Native (Bare-Metal) vs Hosted Hypervisors