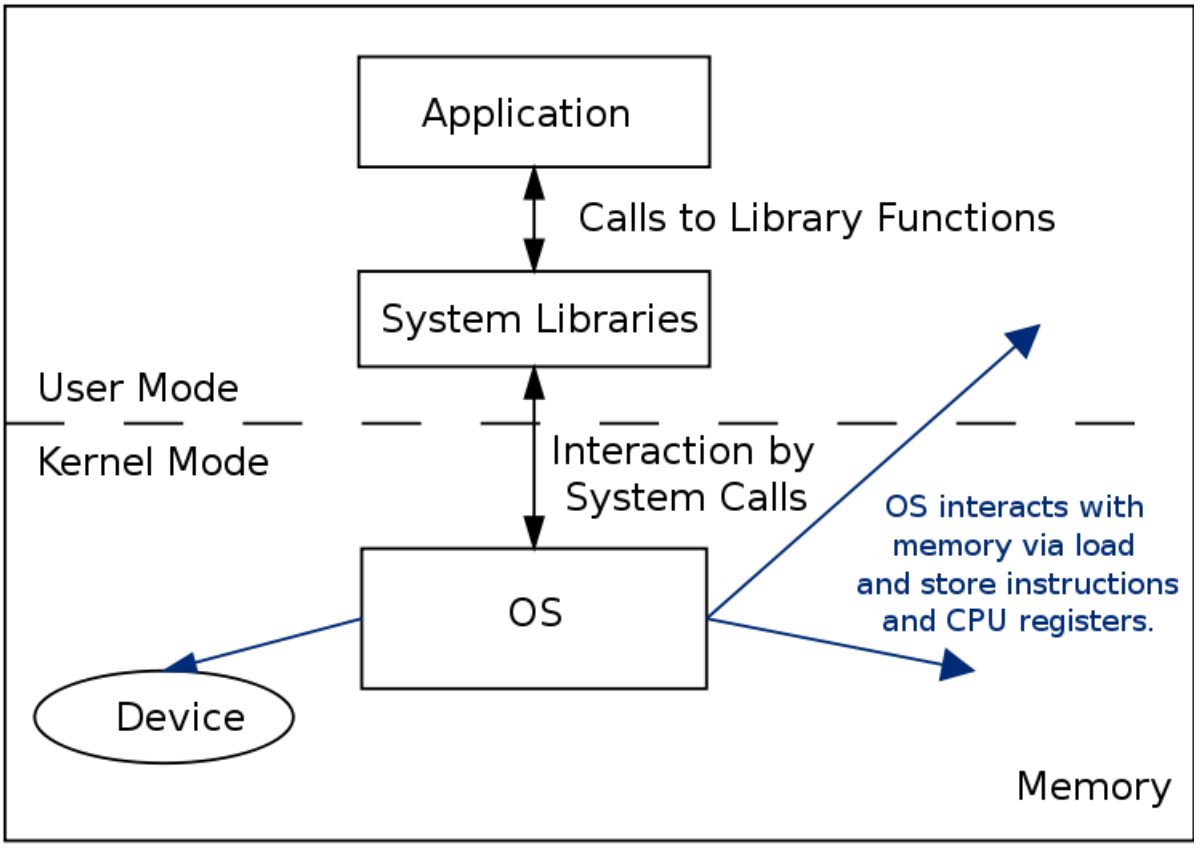
Operating Systems Notes

Here are are some rough notes I put together as part of revision for a uni course. They are heavily based on the course lecture notes by Kevin Elphinstone and Leonid Ryzhyk. ***All diagrams****are sourced from those lecture notes, some of which are in turn from the text book, A. Tannenbaum, Modern Operating Systems.*

OS Overview

- The OS **hides the details of the hardware** and provides an abstraction for user code.
- The OS is responsible for **allocating resources** (cpu, disk, memory…) to users and processes. It must ensure no starvation, progress, and that the allocation is efficient.
- The kernel is the portion of the OS running in privileged mode (hardware must have some support for privileged mode for the OS to be able to enforce the user and kernel mode distinction).



- When the hardware is in privileged mode all instructions and registers are available. When the hardware is in user mode only a limited sets of instructions, registers and memory locations are available. For instance when in user mode, it is not possible to disable interrupts because otherwise a user could ensure that the OS never gets control again..
- A process is a instance of a program in execution.
- A process in memory usually has at least three segments. The **text segment** for the program code (instructions), a **data segment** called the heap for global variables and dynamically allocated memory, and a **stack segment** used for function calls.
- A process also has an **execution context** which includes registers in use, program counter, stack pointer, etc.

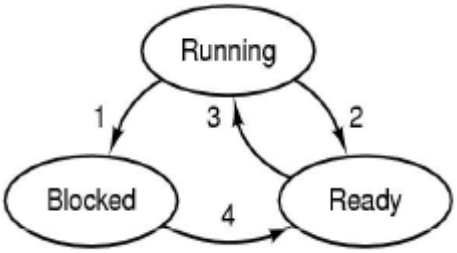
- In a Monolithic OS different parts of the OS like processor allocation, memory management, devices and file systems are not strictly separated into separate layers but rather intertwined, although usually some degree of grouping and separation of components is achieved.

System Calls

- The system call interface represents the abstract machine provided by the operating system.
- man syscalls
- Syscall numbers
- hardware syscall instruction

Processes and Threads

- Processes can contain one or more threads. These threads are units of execution, and always belong to a process.
- A process can terminate by,
  - normal exit (voluntary), usually returning EXIT_SUCCESS
  - error exit (voluntary), usually returning EXIT_FAILURE
  - fatal error (involuntary), eg. segfault, divide by zero error
  - killed by another process (involuntary), eg. pkill (sending the SIGTERM signal)
- Thread states



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- Running to ready could be from a voluntary yield, or end of time allocated by the CPU, so the scheduler moves the thread to the Ready queue to give another process a go.
- Running to blocked could be from a calling a syscall and waiting for the response, waiting on a device, waiting for a timer, waiting for some resource to become available, etc.
- The **scheduler** (also called the dispatcher) decides which thread to pick from the read queue to run.

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

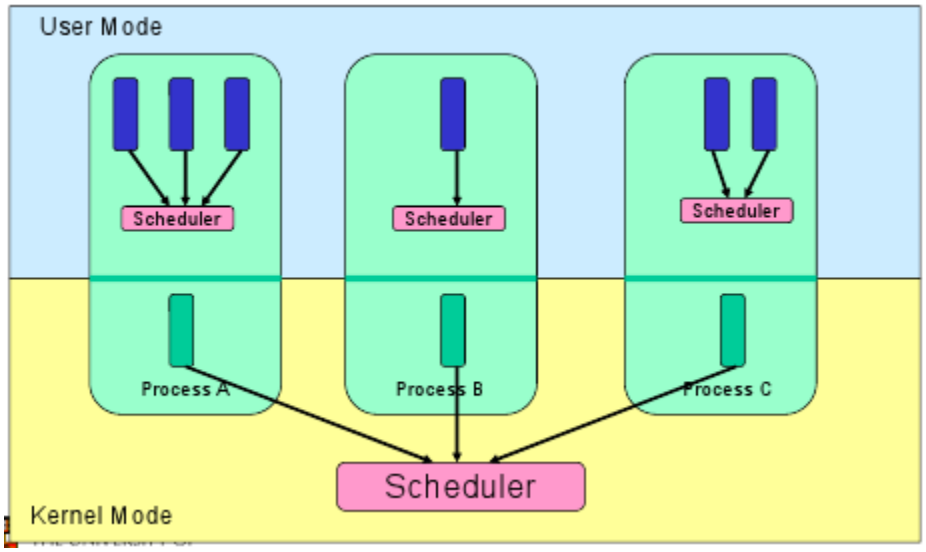
- Items shared by all threads in a process
- Items private to each thread

- The idea is that a process with multiple threads can still make process when a single thread blocks, as if one thread blocks, the others can still be working. In more recent times, it allows a process to have threads running on multiple CPU's on modern multi-core machines.

- The other model is **finite-state machine** where syscall's don't block but instead allow the program to continue running and then send the process an interrupt when the syscall has been dealt with. This is not as easy to program though.
  During the system initialisation background processes (called daemon's on linux, service's on windows) and foreground processes can be started.

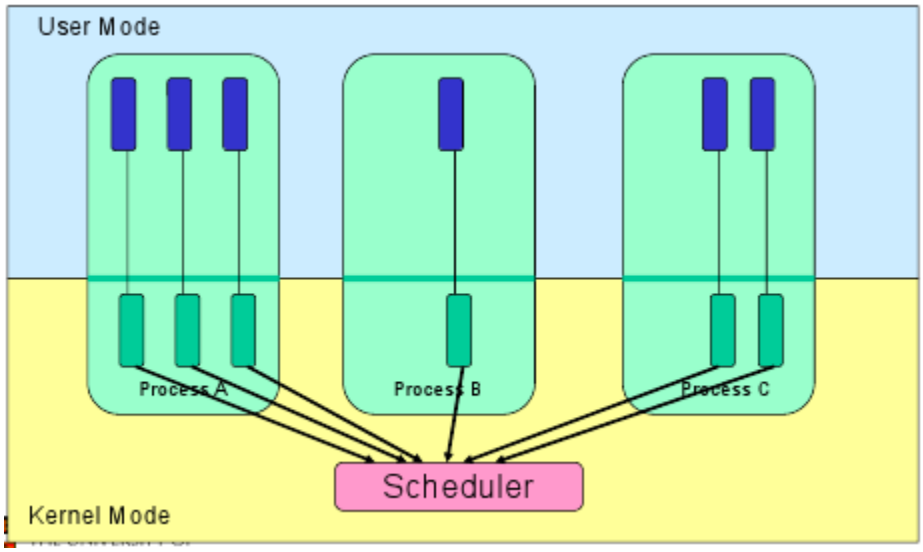
Threads Implementation

- **Kernel-threads vs. User-level threads.**
- User-level threads



  - Thread management is **handled by the process** (usually in a runtime support library)
  - The advantages are,
    - switching threads at user-level is much **cheaper than the OS doing a context switch**
    - you **can tune your user-level thread scheduler**, and not just use the OS provided one
    - **no OS support for threads needed**
  - The disadvantages are,
    - your threads must yield(), you can't really on an interrupt to give your scheduler control again (this is known as co-operative multithreading)
    - cannot take advantage of multiple CPU's
    - if one of your user-level threads gets blocks by the OS, your whole process gets blocked (because the OS only sees one thread running)

- Kernel-level Threads

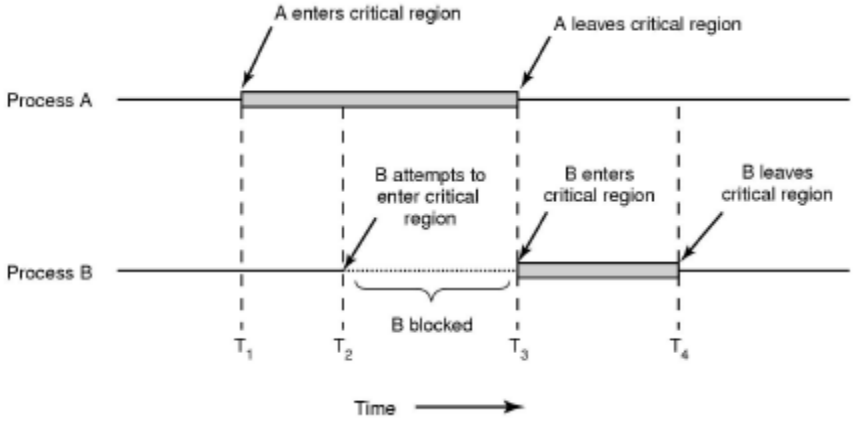


  - Thread management is done by the kernel through system calls
  - The downside is thread management (creation, destruction, blocking and unblocking threads) requires kernel entry and exit, which is expensive
  - The upside is we now have preemptive multithreading (the OS just takes control when it wants to schedule another process, so no need to rely on the thread to yield), can take advantage of a multiprocessor, and individual threads can have isolated blocking.
- A Thread switch can happen in between execution of any two instructions, at the same time it must be transparent to the threads involved in the switch. So the OS must save the state of the thread such as the program counter (instruction pointer), registers, etc. as the thread context. The switch between threads by the OS is called a **context switch**.

Concurrency

- A race condition occurs when the computation depends on the relative speed of two or more processes.
- Dealing with race conditions,
  - **Mutual exclusion**
    - Identify the shared variables,
    - identify the code sections which access these variables (critical region)
    - and use some kind of mutual exclusion (such as a lock) to ensure that at most only one process can enter a critical region at a time.
  - **Lock-free data structures**
    - Allow the concurrent access to shared variables, but design data structures to be designed for concurrent access
  - **Message-based communication**
    - Eliminate shared variables and instead use communication and synchronisation between processes.

- Mutual Exclusion – enter_region() and leave_region().



- Hardware usually provides some mutual exclusion support by providing an **atomic test and set instruction**.
- A uniprocessor system runs one thread at a time, so concurrency arises from preemptive scheduling. But with multiprocessor machines concurrency also arises from running code in parallel using shared memory.
- The problem with the approach to mutual exclusion so far is that when process B is blocked, it just sits in a loop waiting for the critical region to become available. Can fix this by using sleep and wakeup. We use a **mutex** lock for this.
- Blocking locks can only be implemented in the kernel, but can be accessed by user-level processes by system calls.
- **A mutex allows at most one process to use a resource, a semaphore allows at most N processes.**
- **Producer-consumer problem**
  - Producer can sleep when buffer is full, and wake up when space available.
  - Consumer can sleep when buffer is empty and wake up when some items available.
  - Can do using semaphores or condition variables.
- **Monitors** are set up to only allow one process/thread to operate inside it at once, with extra requests put in a queue. Implemented by the compiler.
- **Condition variables**. cv_create, cv_destroy, cv_wait, cv_signal, cv_broadcast
- **Dining philosophers problem**. Need to prevent deadlock.

Deadlock

- 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.
- Four conditions for deadlock:
  - **Mutual exclusion condition** (device not shareable amongst threads)
  - **Hold and wait condition** (a resource can be held, but then block awaiting more resources)
    - Can attack this by requiring all process ask for all resources at the start, and only start if they are granted.
  - **No preemption condition** – previously granted resources cannot be forcibly taken away
  - **Circular wait condition**
    - Always request resources in the same order
- Dealing with deadlock:
  - ignore the problem
  - detect and recover

- dynamic avoidance (careful resource allocation) – we require some information in advance like which resources and how many will be needed before process starts.
  - Bankers algorithm
- prevention (negate one of the four conditions for deadlock)

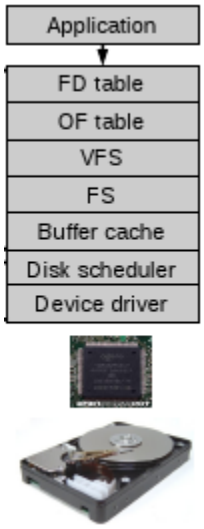
- Starvation is slightly different to deadlock as the system can be making progress, but there are some processes which never make progress.

File Systems

- File systems provide an abstraction of the physical disk. They allow you to store files in a structured manner on a storage device.
- The file system abstraction,

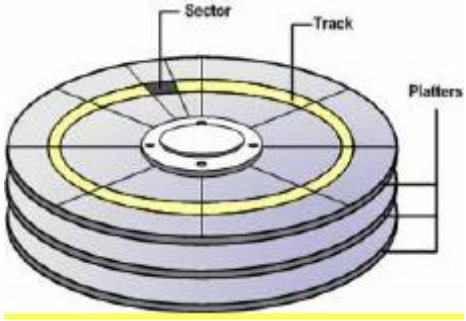
User's view	Under the hood
Hierarchical structure	Flat address space
Arbitrarily-sized files	Fixed-size blocks
Symbolic file names	Numeric block addresses
Contiguous address space inside a file	Fragmentation
Access control	No access control
(Some degree of) reliability	Data written to the disk survives OS crashes. RAID provides additional protection against disk crashes.

- Architecture of the OS storage stack,



- The application interacts with the system call interface which on linux provides creat, open, read, write, etc.

- In the case of modern hard drives,



- The disk controller (between device driver and physical disk) hides disk geometry and exposes a linear sequence of blocks.
- The device driver hides the device specific protocol to communicate with the disk.
- The disk scheduler takes in requests coming from different processes and schedules them to send one by one to the device driver.
- The buffer cache keeps blocks in memory to improve read and write times for frequently accessed blocks.
- The file system (FS) hides the block numbers and instead exposes the directory hierarchy, files, file permissions, etc. To do this it must know which blocks relate to which file, and which part of the file. It must also manage how to store this on the blocks of the linear address space of the disk, keeping track of which blocks are in use. See Allocation Strategies
- The virtual file system (VFS) provides an interface so that different file systems which are suitable for VFS (ie. they have the concept of files, directories, etc..) can be accessed uniformly.
- The open file table (OF table) and file descriptor table (FD table) keep track of files opened by user-level processes.
  There are many popular file systems in use today. FAT16, FAT32 are good for embedded devices; Ext2, Ext3, Ext4 are designed for magnetic disks, ISO9660 is designed for CD-ROM's, JFFS2 is optimised for flash memory as you don't want to write to the same location too many times as it wears out. Others include NTFS, ReiserFS, XFS, HFS+, UFS2, ZFS, JFS, OCFS, Btrfs, ExFAT, UBIFS.

Performance issues with standard magnetic disks

- To access a block on a disk the head must move to the required track (**seek time**), then we have to wait until the required block on the track reaches the head (**rotational delay**). We also have some mostly fixed overhead for the data to be passed between the device and the driver (**transfer time**).
- Total average access time, $T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$. Where *r* is the rotational speed, *b* is the number of bytes to transfer, *N* is the number of bytes per track and T_s is the seek time.
- Seek time is the most expensive, so we want to ensure that our **disk arm scheduler** gives good performance.
  - **First-in, First-out** (no starvation)
  - **Shortest seek time first** (good performance, but can lead to starvation)
  - **Elevator (SCAN)** (head scans back and forth over the disk. no great starvation, sequential reads are poor in one of the directions)
  - **Circular SCAN** (head scans over the disk in one direction only)

Block Allocation Strategies

- **Contiguous Allocation**
  - Gives good performance for sequential operations as all the blocks for a file are together in order
  - though you need to know the maximum file size at creation
  - as files are deleted, free space is fragmented (**external fragmentation**)

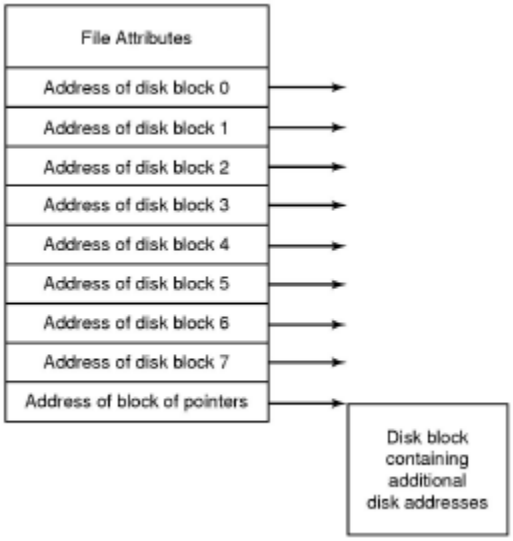
- **Dynamic Allocation**
  - Blocks allocated as needed, hence the blocks for a file could be all over the place on the disk
  - need to keep track of where all the blocks are and the order for each file

    External fragmentation - **space wasted external to the allocated regions**, this space becomes unusable as its not contiguous (so you have lots of small spaces but you need a larger space to fit a whole file in)

    Internal fragmentation – **space wasted internal to the allocated memory regions,** eg. you get a 4K block allocated and only use 1K, but the OS can't give the leftover 3K to someone else.

Keeping track of file blocks

- File allocation table (used for FAT16 and FAT32)
- inode based FS; keep an index node for each file which stores all the blocks of the file.



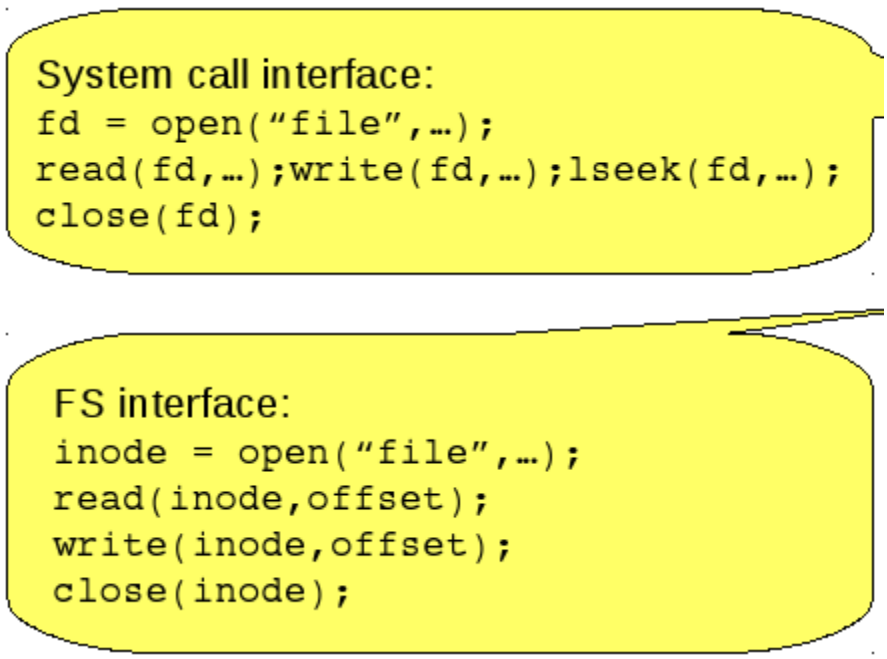
  -
- Directories are stored as normal files but the FS gives these file special meaning.
- A directory file stores a list of directory entries, where each entry containing the file name (because Ext2/3/4 store these with the directory file, not the inode for the file), attributes and the file i-node number.
- Disk blocks (sectors) have a hardware set size, and the file system has a filesystem set block size which is sector size * 2^N, for some N. A larger N means large files need less space for the inode, but smaller blocks waste less space for lots of small files.

- Ext2 inodes

mode
uid
gid
atime
ctime
mtime
size
block count
reference count
direct blocks (12)
single indirect
double indirect
triple indirect

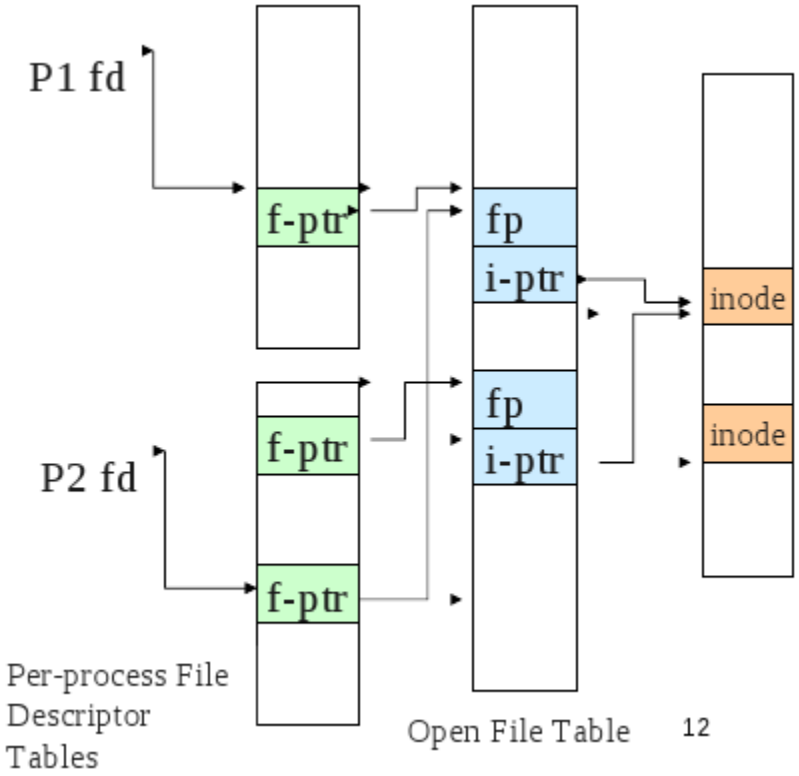
- The top bunch of data is file attributes. Block count is the number of disk blocks the file uses.
- The direct blocks area stores index's for the first 12 blocks used by the file.
- The single indirect is a block numbers to a block which contains more block numbers.w

System call interface



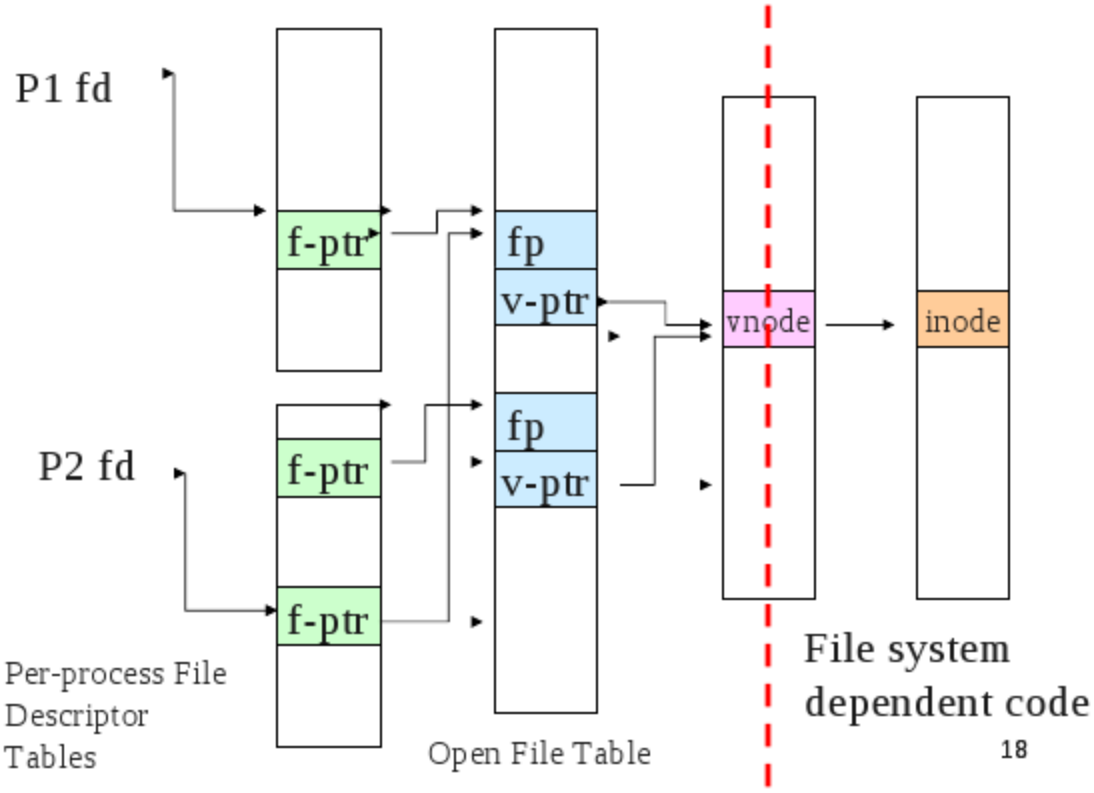
- At the system call level a file descriptor is used to keep track of open files. The file descriptor is associated with the FS inode, a file pointer of where to read or write next and the mode the file was opened as like read-only.

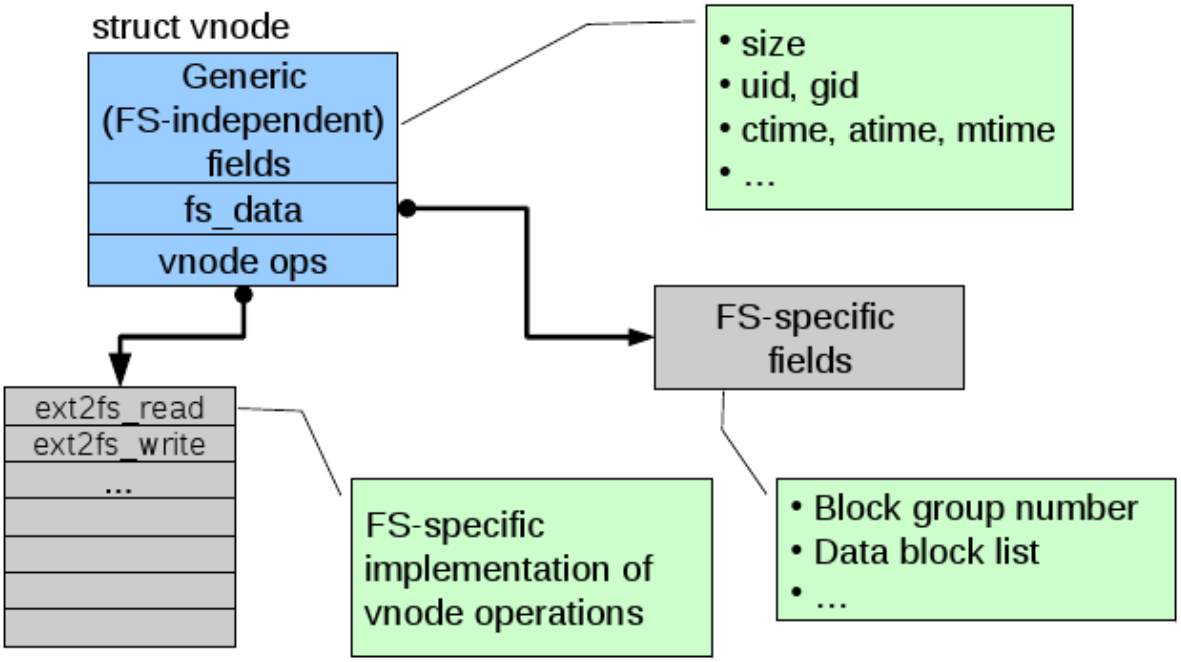
- Most Unix OS's maintain a per-process fd table with a global open file table.



VFS

- Provides a single system call interface for many file systems.
  - the application can write file access code that doesn't depend on the low level file system
  - device can be hard disk, cd-rom, network drive, an interface to devices (/dev), an interface to kernel data structures (/proc)

- 

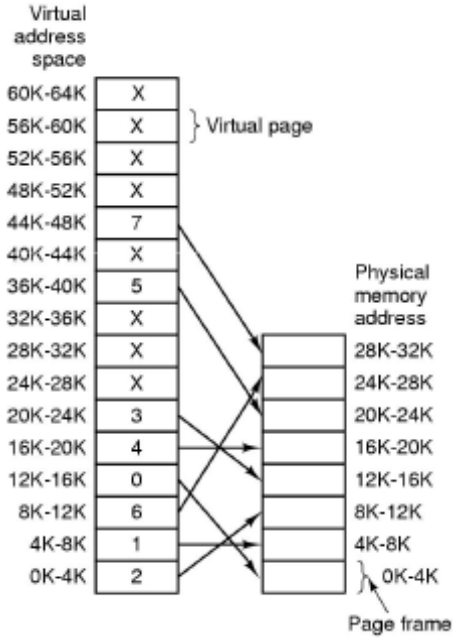
- 

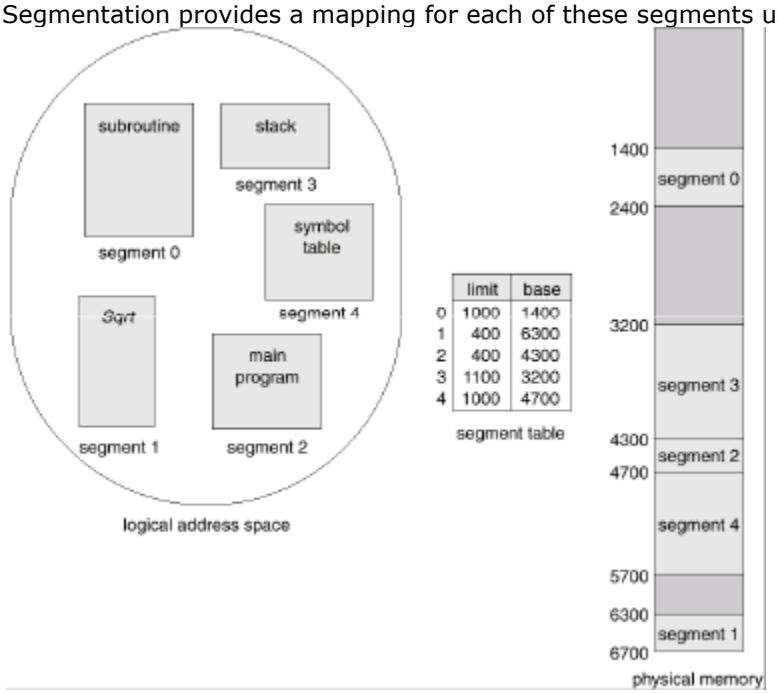
- Journaling file system keeps a journal (or log) of FS actions which allows for the FS to recover if it was interrupted when performing an action that is not atomic, but needs to be.

Memory Management and Virtual Memory

- The OS needs to keep track of physical memory, what's in use and which process is it allocated to. It must also provide applications with a view of virtual memory that they are free to use.

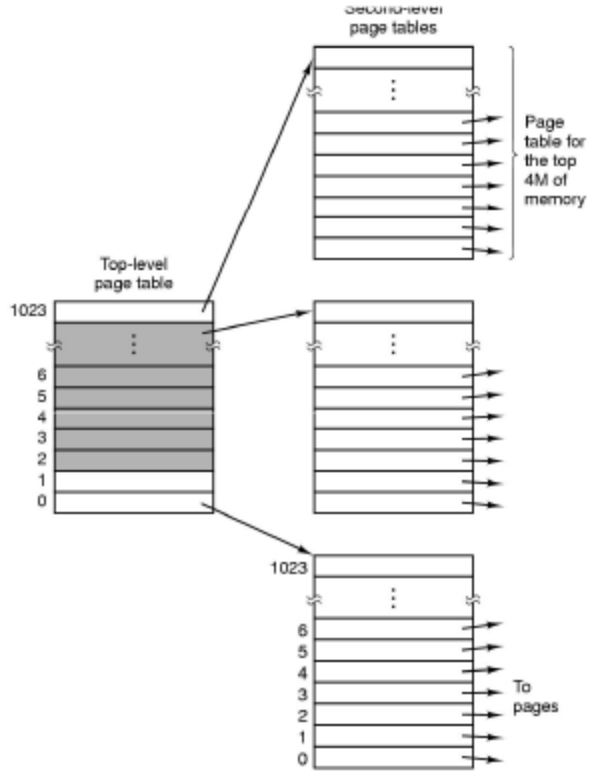
- Swapping (sometimes called paging) is where memory is transferred between RAM and disk to allow for more data in memory than we have space for in physical RAM. On base-limit MMU's swapping only allows who programs memory allocation to be swapped to disk, rather than just pages at a time as with virtual memory, hence you can't use swapping here to allow programs larger than memory to run.
- If we are only running one program we can give it all of memory (and either run the in part of it, or in some other ROM). However many systems need more than one process to be running at the same time. Multiprogramming also means that we can be utilising the CPU even when a process is waiting on I/O (ie. give the CPU to the other process). But to support multiprogramming we need to divide memory up.
- We could divide memory up into fixed size partitions and allocate these to processes, but this creates **internal fragmentation (wasted space inside the partition allocated)**.
- Using dynamic partitioning we give each process exactly how much it needs, though this assumes we know how much memory we need before the process is started. This approach also leads to **external fragmentation where we have lots of little unallocated gaps**, but these are too small to be used individually.
- Approaches to dynamic partitioning include **first-fit**, **next-fit**, **best-fit** and **worst-fit**.
- Binding addresses in programs
  - Compile time
  - Load time
  - Run time
- Protection – we only want each process to be able to access memory assigned to that process, not other process
  - **Base and Limit Registers** - set by the kernel on a context switch, the hardware handles the rest
  - **Virtual Memory** - two variants
    - **Paging**
    - Partition physical memory into small equal sized chunks called **frames**.
    - Divide each process's virtual (logical) address space into same size chunks called**pages**. So a virtual memory address consists of a page number and an offset within that page.
    - OS maintains a **page table** which stores the frame number for each allocated virtual page.



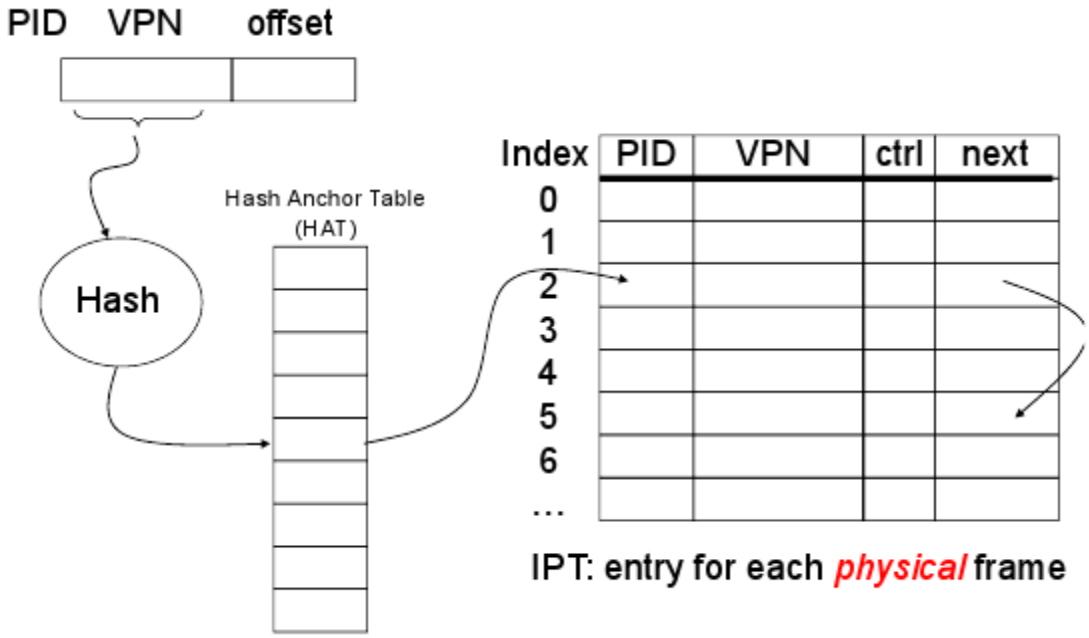
- No external fragmentation, small internal fragmentation.
- Allows sharing of memory
- Provides a nice abstraction for the programmer
- Implemented with the aid of the MMU (memory management unit).
- **Segmentation**
- A program's memory can be divided up into segments (stack, symbol table, main program…)
- Segmentation provides a mapping for each of these segments using a base and limit.
- 

Virtual Memory

- If a program attempts to access a memory address which is not mapped (ie. is an invalid page) a page fault is triggered by the MMU which the OS handles.
- Two kinds of page faults,
  - Illegal Address (protection error) – signal or kill the process
  - Page not resident – get an empty frame or load the page from disk, update the page table, and restart the faulting instruction.
- Each entry in the page table not only has the corresponding frame number but also a collection of bits like protection bits, caching disabled bit, modified bit, etc.
- Page tables are implemented as a data structure in main memory. Most processes don't use the full 4GB address space so we need a data structure that doesn't waste space.
- The **two level page table** is commonly used,

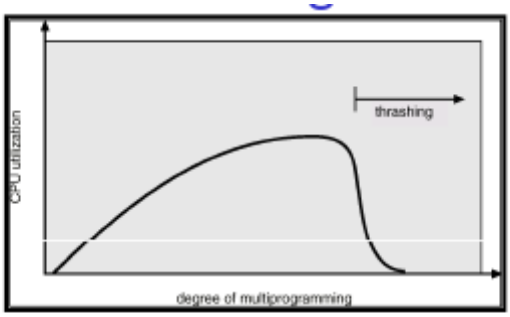


- An alternative is the **inverted page table**,



- The IPT grows with size of RAM, not virtual address space like the two level page table.
- Unlike two level page table which is required for each process, you only need one IPT for the whole machine. The downside is sharing pages is hard.
- Accessing the page table creates an extra overhead to memory access. A cache for page table entries is used called a **translation look-aside buffer (TLB)** which contains frequently used page table entries.

- TLB can be hardware or software loaded.
- TLB entries are process specific so we can either flush the TLB on a context switch or give entries an address-space ID so that we can have multiple processes entries in the TLB at the same time.
- Principle of Locality (90/10 rule)
  - **Temporal Locality** is the principle that accesses close together in terms of time are likely to be to the same small set of resources (for instance memory locations).
  - **Spatial locality** is the principle that subsequent memory accesses are going to be close together (in terms of their address) rather than random. array loop example
- The pages or segments required by an application in a time window (Δ) is called its memory**working set**.
- Thrashing,



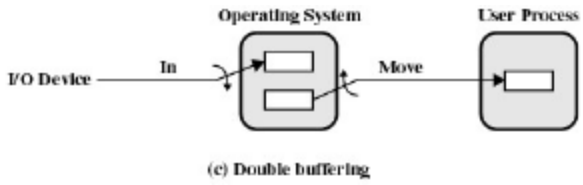
- **Why does thrashing occur?**

Σ working set sizes > total physical memory size

- To recover from thrashing, just suspend some processes until it eases.
- **Fetch policy** – when should a page be brought into memory? on demand, or pre-fetch?
- **Replacement policy** – defines which page to evict from physical memory when its full and you start swapping to disk.
  - **Optimal**
  - **FIFO** – problem is that age of a page isn't necessarily related to its usage
  - **Least Recently Used** – works well but need to timestamp each page when referenced.
  - **Clock** (aka. Second Chance) – an approximation of LRU. Uses a usage or reference bit in the frame table
- **Resident Set Size** – how many frames should each process have?
  - Fixed allocation
  - Variable allocation – give enough frames to result in an acceptable fault rate

Input Output

- **Programmed I/O** (polling or busy waiting)
- **Interrupt Driven I/O**
  - Processor is interrupted when I/O module (controller) is ready to exchange data
  - Consumes processor time because every read and write goes through the processor
- **Direct Memory Access** (DMA)
  - Processor sent interrupt at start and end, but is free to work on other things while the device is transferring data directly to memory.



(c) Double buffering

Scheduling

- The scheduler decides which task to run next. This is used in multiple contexts, multi-programmed systems (threads or processes on a ready queue), batch system (deciding which job to run next), multi-user system (which user gets privilege?).
- Application behaviour
  - **CPU bound** – completion time largely determined by received CPU time
  - **I/O bound** – completion time largely determined by I/O request time
- Preemptive (requires timer interrupt, ensures rouge processes can't monopolise the system) v non-preemptive scheduling.
- Scheduling Algorithms can be categorised into three types of systems,
  - **Batch systems** (no users waiting for the mouse to move)
  - **Interactive systems** (users waiting for results)
    - Round Robin – each process is run and if it is still running after some timeslice t, the scheduler preempts it, putting it on the end of the ready queue, and scheduling the process on the head of the ready queue.
      If the timeslice is too large the system becomes sluggy and unresponsive, if it is too small too much time is wasted on doing the context switch.
    - The traditional UNIX scheduler uses a priority-based round robin scheduler to allow I/O bound jobs to be favoured over long-running CPU-bound jobs.
  - **Realtime systems** (jobs have deadlines that must be meet)
    - Realtime systems are not necessarily fast, they are **predictable**.
    - To schedule realtime tasks we must know, its arrival time a_i, maximum execution time (service time), deadline (d_i).
    - tasks could be periodic or sporadic.
    - slack time is time when the CPU is not being used, we are waiting for the next task to become available.
    - two scheduling algorithms,
    - **Rate Monotonic** – priority driven where priorities are based on the period of each task. ie. the shorter the period the higher the priority
    - **Earliest Deadline First** – guaranteed to work if set of tasks are schedulable. The earlier the deadline the higher the priority.

Multiprocessor Systems

- For this section we look at shared-memory multiprocessors.
- There are uniform memory accesses multiprocessors and non-uniform memory access multiprocessors which access some parts of memory slower than other parts. We focus on the UMA MP's.
- "**Spinlocking** on a uniprocessor is useless, as another thread on the same processor needs to release it, so blocking asap is desirable. On a multiprocessor, the thread holding the lock may be presently active on another processor, and it could release the lock at any time.
  On a multiprocessor, spin-locking can be worthwhile if the average time spent spinning is less than the average overheads of context switch away from, and back to, the lock requester."

- **Thread affinity** refers to a thread having some preferred processor to run on an a multiprocessor machine. For instance if thread A started on cpu0 it may want to be scheduled again on cpu0 rather than cpu1 as parts of the cache may still be intact.
- **Multiprocessor systems have multiple ready queues**, as just one ready queue would be a shared resource which introduces **lock contention**.

Source: http://andrewharvey4.wordpress.com/2010/07/31/operating-systems-notes/