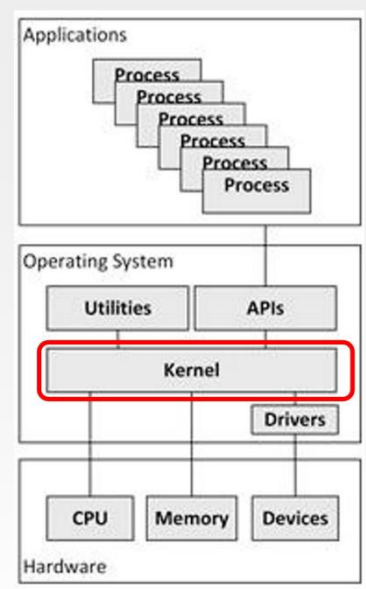
INF113 KEY TOPICS

Introduction to OS:

the components of an OS.

For components of an operating system:



1. Kernel
    - It is a computer program and the heart of an operating system.
    - It is one of the first programs to load into main memory when you turn on a computer.
    - Its primary function is to manage all hardware resources.
    - Most importantly, the kernel schedules all the access to hardware to avoid conflicts if multiple programs try to access the same resource or device simultaneously.
2. Utilities
    - They are also called utility software
    - Designed to analyze, configure, optimize or maintain a computer
    - They are used to support computer infrastructure – in contrast to application software.
    - Examples: backup software, network managers, disk repair tools, file compression.
3. APIs
    - Application Programming Interface

- Allow user processes to request a service from the kernel, such as opening a file from a hard disk or reading data from memory.
- A user does so by making a system call, e.g. read(), write(), open()

4. Drivers
    - Each of them is a program that operates or controls a particular hardware device.
    - They provide an interface for the kernel to access hardware devices without needing to know details about the devices.
    - Many device drivers are built into an OS and become part of the OS. If a user buys a new type of device, the new device driver might need to be installed by the user.

what does the OS do?

- Allows many programs to run at the same time (CPU, scheduling)
- Allows programs to share memory
- Enables programs to interact with the input/output (I/O) devices

The OS is in charge of making sure that the system operates correctly and efficiently in an easy-to-use manner.

Important function of an OS

- Processor management
    - decides which process can run on CPU at what time and for how long
- Memory management
    - decides which process can get memory at what time and how much
- I/O Device management
    - decides which process can get an I/O device at what time and for how long
- Communication management
    - manages the communication between different processes
- File management
    - manages and control how data is stores and retrieved
    - provides interfaces for users to create, move, or delete files and directories
- Job accounting
    - keep tracking of time and resources used by various jobs and users
- Security
    - provides authentication features for each user by means of passwords
- Protection
    - protects various processes from each other's activities

    - ensures that all access to system resources is controlled
- Error handling
    - keeps checking for possible errors
    - takes an appropriate action to ensure correct and consistent computing

user mode vs. kernel mode

User mode

- Code that runs in user mode is restricted in what it can do. For example, when running in user mode, a process can't issue I/O requests.

Kernal mode

- The operating system (or kernel) runs in this mode. In this mode, code that runs can do what it likes, including privileged operations such as issuing I/O requests and executing all types of restricted instructions.
- The superuser can access all files regardless of privileges.

The dual mode of operation provides us with the means for protecting the operating system from errant users.

What a user process needs to do to perform privileged operations:

- Invoke corresponding system calls.
- System calls allow the kernal to carefully expose certain key pieces of functionality to user programs, such as
    - open(), malloc(), fork(), etc
- To execute a system call, a program must execute a special trap instruction (which switches the system to kernal mode and jumps into the OS to a pre-specified destination: the trap table). When finished, the OS calls a special return-from-trap instruction (which switches the system to user mode) and passes control back to the user program.

Other means of entering the kernal mode: exception and interrupts (e.g. timer interrupt).

All of this is a part of the limited direct execution protocol which runs programs efficiently but without loss of OS control.

different kinds of OS interfaces

User and OS interface

- There are several ways for users to interface with the OS. Three fundamental approaches are:
    - Graphical user interface (GUI)
    - Touch screen
    - Command line interface, e.g. the bash shell command interpreter in macOS.

Command line interface (command interpreter)

- The command interpreters are known as shells, e.g. bash
- The main function of the command line interface is to get and execute the next user-specified command
- Many of the commands given at this level manipulate files

Virtualization of CPU:

Physical resources are limited

- Processor (CPU)
- Memory
- Disk

Virtualization of CPU and memory → limited resources seem unlimited

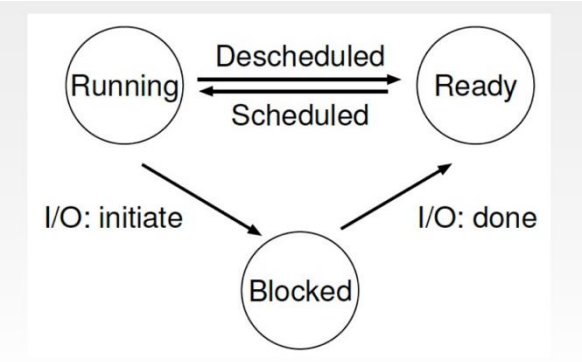
The OS provide the illusion of a nearly endless supply of said CPUs by virtualizing the CPU.

- By running one process, the stopping it and running another, and so forth.
- Cons of time-sharing: performance as each process will run more slowly if the CPU(s) must be shared.

Process

- A process is a running program
- A process is the abstraction provided by the OS of a running program.
- Components of machine state that comprises a process:
  - Memory
  - Registers
  - I/O information

Process states:



To implement the virtualization of the CPU well, the OS needs

- Mechanisms: low-level machinery
  - How to switch between processes?
- Policies: high-level intelligence
  - Which process to choose next
  - Policies are algorithms for making some kind of decision within the OS.

fork(), exec(), wait()

System calls for:

- process creation
    - fork(): starts a new process (a child process) which is a copy of the main process. Both parent and child processes are executed simultaneously in case of fork().
    - excec(): replaces the current process image with a new one. Control never returns to the original program unless there is an exec() error.
- a process to wait for a process it has created to complete
    - wait()

timer interrupts

When a process is running on the CPU, this by definition means the OS is not running. If the OS is nor running, it cannot take an action to switch the process.

Approaches to regain control of the CPU:

- A cooperative approach: OS is passive
    - A process gives up the CPU when it
        - Accesses I/O
        - Creates a new process
        - Sends a message to another machine
        - Executes an illegal operation
    - The OS trusts the processes of the system to behave reasonably.
    - In a cooperative scheduling system, the OS regains control of the CPU by waiting for
        - A system call
        - An illegal operation to take place
- A non-cooperative approach: OS is in control (timer interrupt)

Timer interrupt

- The OS sets the timer, and when the timer reaches 0, an interrupt occurs.
- When the interrupt is raised, the currently running process is halted, and a pre-configured interrupt handler in the OS runs (the OS has regained control of the CPU)
- This hardware feature is essential in helping the OS maintain control of the machine.

context switching

Process control block (PCB)

- Each process is represented in the OS by a process control block.
- The PCB serves as the repository for all the data needed to start, or restart, a process, along with some accounting data.

Context switching

- The OS saves the PCB of the currently running process, and then restore the PCB for the soon-to-be-executing process.
- By doing so, the OS ensures that when the return-from-trap or return-from-interrupt instruction is finally executed, instead of returning to the process that was running, the system resumes execution of another process.
- This is a low-level technique.

time slices

Round Robin is sometimes called time-slicing since RR runs a job for a time slice and then switches to the next job in the run queue.

The length of the time slice is critical in RR:

- The shorter it is, the better the performance og RR under the response-time metric.
- However, making the time slice too short is problematic:
    - Suddenly the cost of context switching will dominate overall performance.
    - Deciding on the length of the time slice presents a trade-off to a system design.

Amortizing can reduce costs.

what does the scheduler do?

- The scheduler decides which jobs to run first and for how long.
- Scheduling is a high-level policy

scheduling policies

We have two scheduling metrics: turnaround time and fairness

- Performance and fairness are often at odds in scheduling.

Metric: Turnaround time

- First In, First Out (FIFO). Simple and easy to implement.
- Shortest Job First (SJF)
- Shortest Time-to-Completion First (STCF)
    - Any time a new job enters the system, the STCF scheduler determines which of the remaining jobs (including the new job) has the least time left, and then schedules that one.
- SJF and STCF optimizes turnaround time but is bad for response time.

Metric: Response time

- Round Robin (RR), sometimes called time-slicing
    - Instead of running jobs to completion, RR runs a job for a time slice and then switches to the next job in the running queue. It repeatedly does so until the jobs are finished.
    - RR optimizes response time but is bad for turnaround time.

how to schedule without a priori knowledge of job length?

Multi-level feedback queue (MLFQ)

- A system that learns from the past to predict the future.
- Trick: Since the OS doesn't know whether a job will be a short job or a long-running job, it first assumes the job might be a short job, thus giving the job high priority.

MLFQ rules:

Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).

Rule 2: If Priority(A) = Priority(B), A & B run in round-robin fashion using the time slice (quantum length) of the given queue.

Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue).

Rule 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).

Rule 5: After some time period S, move all the jobs in the system to the topmost queue.

turnaround time, response time

Turnaround time:

- It is a scheduling metric concerning performance

$$T_{turnaround} = T_{completion} - T_{arrival}$$

Response time:

- It is another scheduling metric concerning fairness

$$T_{response} = T_{firstrun} - T_{arrival}$$

The utilization of CPU

Utilizing the processor

- By treating each CPU burst as a job, the scheduler makes sure processes that are "interactive" get run frequently.
- While those interactive jobs are performing I/O, other CPU-intensive jobs run, thus better utilizing the processor.

scheduling-related security

How to prevent gaming of our scheduler

- Perform better accounting of CPU time at each level of the MLFQ.
- Instead of forgetting how much of a time slice a process used at a given level, the scheduler should keep track.
- Once a process has used its allotment, it is demoted to the next priority queue. Whether it uses the time slice in one long burst, or many small ones does not matter.

With gaming tolerance

- With such protection in place, regardless of the I/O behavior of the process, it slowly moves down the queue, and thus cannot gain an unfair share if the CPU.

Virtualization of Memory:

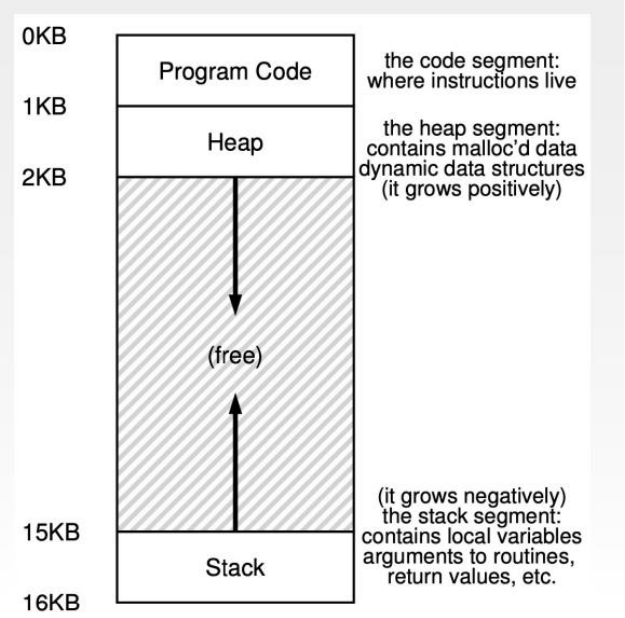
address space

- An easy to use abstraction of physical memory
- Multiple processes can share the memory

Address space

- It is the running program's view of memory in the system.
- The address space of a process contains all of the memory state of the running program.
    - Code
    - Heap
    - Stack

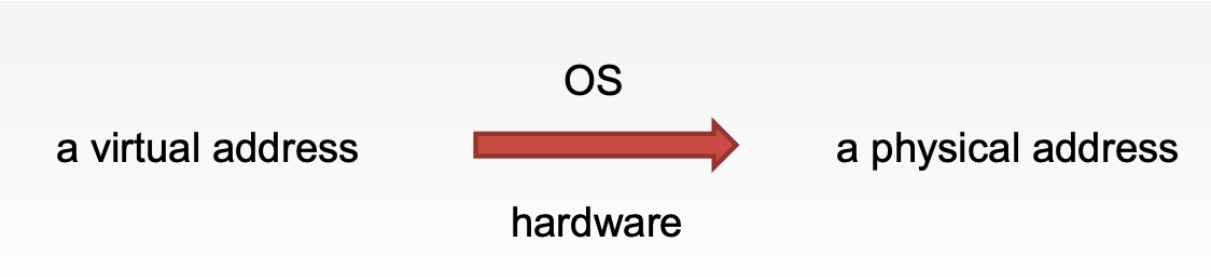
An example of an address space:



virtual address, physical address

- The program is not really in memory at physical address 0 through 16KB; rather it is located at some arbitrary physical address(es).
- The illusion:
    - Each program has its own private memory, where its own code and data resides.

- The reality:
    - Many programs are actually sharing memory at the same time.



Goals

- Transparency
    - The OS should implement virtual memory in a way that is invisible to the running program.
    - Thus, the program shouldn't be aware of the fact that memory is virtualized; rather, the program should behave as if it has its own private physical memory.
- Efficiency
    - The virtualization should be as efficient as possible, both in terms of time and space
    - Therefore, the OS will have to rely on hardware support, including hardware features such as TLBs.
- Protection
    - The OS should make sure to protect processes from one another as well as the OS itself from processes.
    - When one process performs a load, a store, or an instruction fetch, it should not be able to access or affect in any way the memory contents of any other process or the OS itself (that is, anything outside its address space).

address translation

Address translation

- Transforming a virtual address into a physical address
    - The hardware takes the virtual address the process thinks it is referencing and transforms it into a physical address which is where the data actually resides.
        - Physical address = base + virtual address

base and bound register

Base and bounds

- Two hardware registers within each CPU:
  - The base register
  - The bounds register (sometimes called a limit register)
- This base-and-bounds pair is going to allow us to
  - place the address space anywhere we'd like in physical memory
  - ensure that the process can only access its own address space
- In this setup, each program is written and compiled as if it is loaded at address zero. However, when a program starts running, the OS decides where in physical memory it should be loaded and sets the base register to that value.

Protection

- The processor will check that the memory reference is within bounds to make sure it is legal
- The bounds register is there to help with protection

Bound register

- If a process generates a virtual address that is greater than the bounds, or one that is negative, the CPU will raise an exception, and the proves will likely be terminated.
- Two ways to define the bound registers, here for simplicity we assume:
  - It holds the size of the address space, and thus the hardware checks the virtual address against it first before adding the base.

Registers can only be modified in kernal mode.

Exception handlers

- The CPU must be able to generate exceptions in situations where
  - a user program tries to access memory illegally.
  - a user program tries to change the values of the (privileged) base and bound registers.

Since there are only one base and bounds register pair on each CPU, and their values differ for each running program, the OS must save and restore the base-and-bounds pair when it switches between processes.

memory isolation

The principle of isolation

- Isolation is a key principle in building reliable systems.
- If two entities are properly isolated from one another, this implies that one can fail without affecting the other.
- Operating systems strive to isolate processes from each other and in this way prevent one from harming the other.
- By using memory isolation, the OS further ensures that running programs cannot affect the operation of the underlying OS.
- Some modern OS's take isolation even further, by walling off pieces of the OS from other pieces of the OS: This provides greater reliability.

memory related errors (for example: segmentation fault, buffer overflow)

Segmentation fault

- Segmentation:
    - Instead of having just one base and bounds pair in our MMU (Memory Management Unit), why not have a base and bounds pair per logical segment (program code, heap and stack) of the address space.

- When we try to refer to an illegal address
    - The hardware detects that the address is out of bounds, traps into the OS, likely leading to the termination og the offending process.

Top two bits	Which segments
00	Code
01	Heap
10	-
11	Stack

Buffer overflow

- Not allocating enough memory

Other memory related errors

- Forgetting to initialize allocated memory
    - You call malloc() properly, but forget to fill in some values into your newly-allocated data type
    - Your program will encounter an uninitialized read, where it reads from the heap some data of unknown value. This unknown value can be something random and harmful.
- Forgetting to free memory
    - Memory leak occurs.
    - Leaking memory (in long-running applications or systems) eventually leads one to run out of memory.
- Freeing memory before you are done with it
    - Called dangling pointer (a pointer points to nowhere)
- Freeing memory repeatedly
    - Programs also sometimes free memory more than once; this is known as the double free.
    - The result of doing so is undefined.
    - The memory-allocation library might get confused and do all sorts of weird things; crashes are a common outcome.
- Calling free() incorrectly
    - free() expects you only to pass to it one of the pointers you received from malloc() earlier.
    - When you pass in some other value, bad things can (and do) happen.

Tool

- Valgrind: excellent at helping you locate the source of your memory-related problems.

stack vs. heap

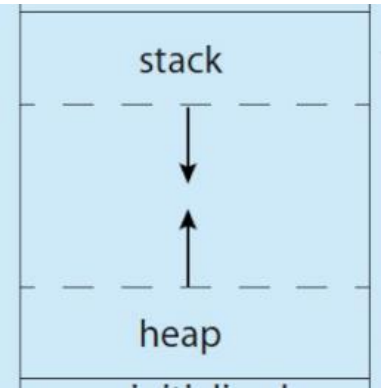
Stack

- Allocations and deallocations of it are managed implicitly by the compiler for the programmer. For this reason it is sometimes called automatic memory.
- C programs use the stack for
    - Function parameters
    - Local variables
    - Return addresses
- The OS allocates this memory and gives it to the process

- The OS will also likely initialize the stack with arguments; specifically, it will fill in the parameters to the main() function, i.e., argc and the argv array for the C programs.
- Stack is LIFO (Last In First Out): the last element which is put on top of the stack becomes the first one being removed from the stack later.

Heap

- The heap is used for explicitly requested (by the programmer) dynamically-allocated data; programs requests such space by calling malloc() and free such space by calling free().
- The heap is needed for data structures, such as linked lists, hash tables, trees, that can change the size during execution.
- The heap will be small at first; as the program runs, and requests more memory via the malloc() library API, the OS may get involved and allocate more memory to the process to help satisfy such calls.
- The use of heap may cause many bugs.
- malloc() returns a pointer to type void. The programmer further helps out by using what is called cast; for example, the programmer can cast the return type of malloc() to a pointer to an int (int *)
- To free heap memory that is no longer in use, programmers simply call free().
- malloc() is a library call
- when the heap runs out of space: malloc() returns NULL



free list management (best-fit, worst-fit, first-fit, next-fit)

Splitting of free space

- When a request is smaller than the free space chunk: the allocator will find a free chunk of memory that can satisfy the request and split it into two. One chunk to the caller, and one will remain on the list.

best-fit:

- Which returns the one closest in size that satisfies the desired allocation to the requester.
- Best-fit is also called smallest fit
- Best fit tries to reduce wasted space. However, it pays a heavy performance penalty when performing an exhausting search for the correct block.

worst-fit:

- Which returns the largest one and keeps the remaining part in the free list.
- Worst-fit tries to leave big chunks free instead of lots of small chunks that can arise from a best-fit approach.
- Since this approach also require a full search of free space, it is costly.

first-fit:

- Which returns the first one in the free list that is big enough.
- First-fit has the advantage of speed.
- But first-fit may pollute the beginning of the free list with small objects.
    - Solution 1: address-based ordering:
        - By keeping the list ordered by the address of the free space, coalescing becomes easier, and fragmentation tends to be reduced.
    - Solution 2: next-fit

next-fit

- Instead of always beginning the first-fit search at the beginning of the list, the next fit algorithm keeps an extra pointer to the location within the list where one was looking last. The idea is to spread the searches for free space throughout the list more uniformly, thus avoiding splintering of the beginning of the list.
- The performance is quiet similar to first-fit, as an exhaustive search is once again avoided.

internal fragmentation vs. external fragmentation

Internal fragmentation

- Fragmentation within the address space (unnecessary free space)
- This simple approach of using a base and bounds register pair to virtualize memory is wasteful.

- It also makes it quite hard to run a program when the entire address space doesn't fit into memory.

External fragmentation

- The free space gets chopped into little pieces of different sizes and is thus fragmented.
- When physical memory quickly becomes full of little holes of free space, making it difficult to allocate new segments, or to grow existing ones.
- Example: When a request to grow a segment arrives, if the next so many bytes of physical space are not available, the OS will have to reject the request, even though there may be free bytes available elsewhere in physical memory.
- Solution: free list management.

coalescing

The problem we need to solve when adding free space back into our list without too much thinking, we might end up with a list seemingly divided into chunks of bytes, even though the entire heap might be free.

Coalescing of free space

- What allocators do in order to avoid this problem is to coalesce free space when a chunk of memory is freed.
    - When returning a free chunk in memory, look carefully at the addresses of the chunk you are returning as well as the nearby chunks of free space; if the newly-freed space sits right next to one (or two, as in this example) existing free chunks, merge them into a single large free chunk.
    - Go through the list and merge neighboring chunks; when finished, the heap will be whole again.
- With coalescing, an allocator can better ensure that large free extents are available for the application.

paging

Paging

- Instead of splitting up a process's address space into some number or variable-sized logical segments, we divide it into fixed-sized units, each of which we call a page.

- Correspondingly, we view physical memory as an array of fixed-sized slots called page frames. Each of these frames can contain a single virtual-memory page.
- Can lead to high performance overheads (paging requires a large amount of mapping information)

Advantage of using paging

- Flexibility
    - With a fully-developed paging approach, the system will be able to support the abstraction of an address space effectively, regardless of how a process uses the address space.
        - We don't need assumptions about the direction of the heap and stack.
- Simplicity
    - The OS keeps a free list of all free pages, and just grabs the first free pages of this list.

Page table

- The page table stores the address translations for each of the virtual pages of the address space and let us know where in physical memory each page resides.
- A data structure used to map virtual addresses (virtual page numbers) to physical addresses (physical frame numbers)
- The simplest form is an array.
- Since page tables can get terribly large, we store the page table for each process in memory somewhere.
- Without careful design of both hardware and software, page tables will cause the system to run slowly, as well as take up too much memory.

code sharing, page sharing

Code sharing

- Saves memory
- While each process thinks that it is accessing its own private memory, the OS is secretly sharing memory which cannot be modified by the process, and thus the illusion is preserved.
- In addition to checking whether a virtual address is within bounds, the hardware also has to check whether a particular access is permissible. If not, the hardware should raise an exception and let the OS deal with the offending process.

Page sharing

- Sharing of code (in binaries, or shared libraries) is useful as it reduces the number of physical pages in use, thus reducing memory overheads.

TLB hit, TLB miss, TLB coverage

What is a cache

- Caches are small, fast memories that (in general) hold copies of popular data that is found in the main memory of the system.
- By keeping frequently accessed data in a cache, the system can make the large, slow memory appear to be a fast one.
- Caches are based on the notion of locality
    - Temporal locality and special locality

Translation-lookaside buffer (TLB)

- Helps speed up address translation
- A TLB is a part of the chip's memory-management unit (MMU), aka a part of the hardware.
- A TLB is a hardware cache of popular virtual-to-physical address translations. It is also called address-translation cache.
- Upon each virtual memory reference, the hardware first checks the TLB to see if the desired translation is held therein. If so, the translation is performed (quickly) without having to consult the page table (which has all translations). TLBs make virtual memory possible.

TLB contents

- A typical TLB might have 32, 64 or 128 entries and be what is called fully associative.
    - Fully associative means that any given translation can be anywhere in the TLB, and that the hardware will search the entire TLB in parallel to find the desired translation.

TLB hit

- The TLB holds the translation.

TLB miss

- When the CPU does not fiend the translation in the TLB.
- Then the hardware will access the page table to find the translation, and, assuming that the virtual memory reference generated by the process is valid and accessible, updates the TLB with the translation. This set of actions are costly. Then the hardware retries the instructions. This time the translation is found in the TLB, and the memory reference is processed quickly.

TLB coverage

- Exceeding the TLB coverage: if the number of pages a program accesses in a short period of time exceeds the number of pages that fit into the TLB, the program will generate a large number of TLB misses, and thus run quite a bit more slowly.
- Solution: to include the support for larger page sizes, the effective coverage of the TLB can be increased (however, it causes internal fragmentation problem)

Temporal locality: the quick referencing of memory items in time.

what is the purpose of using multi-level page table?

Multi-level page tables

- How to get rid of all those invalid regions in the page table instead of keeping them all in memory?
    - Chop up the page table into page-sized units: then, if an entire page of page-table entries (PTEs) is invalid, don't allocate that page of the page table at all.
- This approach is so effective that many modern systems employ it (e.g. x86).
- We make page-table lookups more complicated in order to save valuable memory.

The purpose

- To get rid of all those invalid regions in the page table instead of keeping them all in memory

Advantages of using multi-level page tables

- The multi-level table only allocates page-table space in proportion to the amount of address space you are using; thus, it is generally compact and supports sparse address spaces.
- With a multi-level structure, we add a level of indirection through use of the page directory, which points to pieces of the page table; that indirection allows us to place page-table pages wherever we would like in physical memory.

swapping

Swap space

- Swap space is the space on the disk for moving pages back and forth.
- The OS swap pages out of memory to the swap space and swap pages into memory from the swap space.
- The OS remembers the disk address of a given page.
    - The OS could use the bits in the PTE normally used for data such as the PFN of the page for a disk address.
- The size of the swap space is important, as ultimately it determines the maximum number of memory pages that can be in use by a system at a given time.

The present bit

- If it is set to one, it means the page is present in physical memory. If it is rather on disk, memory it is set to zero.

Page fault: the act of accessing a page that is not in physical memory. The OS page-fault handler takes care of this by transferring the desired page form disk to memory, and perhaps first replace some pages in memory to make room for those soon to be swapped in.

These actions all take place transparently to the process.

The page selection policy

- When to bring a page back into memory
    - Demand paging
    - Prefetching: the OS could guess that a page is about to be used, and thus bring it in ahead of time.

Trashing: when memory is oversubscribed, and the memory demands of the set of running processes simply exceeds the available physical memory.

swapping policies (FIFO, random, Least-recently-used (LRU))

The goal in picking a replacement policy

- Maximize the number of cache hits
    - Cache hit: a page that is accessed is found in memory.
- Minimize the number of cache misses

    - Cache misses: a page that is accessed is not found in memory.

The optimal policy

- Throw out the page that is needed the furthest from now.
- The optimal policy will thus serve only as a comparison point to know how close we are to “perfect”
- However, the future is not generally known

A simple policy: FIFO

- First-in, first-out replacement
- Strength: simple to implement

The random policy

- Picks a random page to replace under memory pressure.
- How Random does depends upon how lucky Random gets in its choices.

FIFO and Random are not good enough

- Problem: it might kick out an important page, one that is about to be referenced again.

Least-Recently-Used (LRU)

- Replaces the least-recently-used page.
- The more recently a page has been accessed, perhaps the more likely it will be accessed again.
- To keep track of this a machine could update, on each page access, a time field in memory.
- As the number of pages in a system grows, this policy is expensive.

Types of locality

- Spatial locality
- Temporal locality
- The assumptions of these types of locality plays a large role in the caching hierarchies of hardware systems.

Approximating LRU: The clock algorithm

- Goal: not repeatedly scanning through all of memory looking for an unused page.

Concurrency:

threads vs. processes

Thread is an abstraction for a single running process.

Process	Thread
Each process has its own address space	Threads of a **process share the same address space** and thus can access the same data.
Each process accesses its own program code, heap and stack.	Each thread of a process **has its own stack** but can access the same program code and heap.
Each process has its own program counter (PC) .	Each thead has its own program counter (PC) .
Each process has its own private set of registers it uses for computation.	Each thread has its own private set of registers it uses for computation.
It is also called a single-threaded process.	

Context switching between processes	Context switching between threads
The address space are different. (i.e., Page table should be switched).	The address space remains the same. (i.e., **There is no need to switch page table, but the register state should be saved and restored**).
The state of a process is saved to the process' Process Control Block (PCB).	The state of a thread is saved to the thread's Thread Contol Block (TCB).

Why use threads

- Parallelism
  - Multiple CPUs (multi-processor): a thread per CPU, each CPU performs a portion of the work.
- To avoid blocking program progress due to slow I/O

    - While one thread in your program waits (i.e., is blocked waiting for I/O), the CPU scheduler can switch to other threads, which are ready to run and do something useful.
    - Threading enables overlap of I/O with other activities within a single program.
- Threads share an address space and thus make it easy to share data. Processes are a sounder choice for logically separate tasks where little sharing of data structures in memory is needed.

Global variables are shared between threads.

Multithreaded applications can spread work across multiple CPUs and thus run faster when given more CPU resources.

race condition

Race condition

- A race condition occurs when the behavior of a program depends on the interleaving of operations of different threads.
- The threads run a race between their operations, and the results of the program execution depends on who wins the race.
- We might get the wrong answer (i.e., context switches occur at untimely points in the execution).
- In fact, we may get a different result each time; this, instead of a nice deterministic computation, we call this result indeterminate, where it is not known what the output will be, and it is indeed likely to be different across runs.

critical section

Critical section

- A critical section is a piece of code that accesses a shared variable (more generally, a shared resource) and must not be concurrently executed by more than one thread.

mutual exclusion

Mutual exclusion

- The property guarantees that if one thread is executing within the critical section, the others will be prevented from doing so.

Atomicity

Atomicity

- The solution:
    - In a single step, did exactly whatever we needed done and thus removed the possibility of an untimely interrupt.
- An atomic instruction:
    - When an interrupt occurs, either the atomic instruction has not run at all, or it has run to completion. There is no in-between state in the middle of the atomic execution.
- However, in the general case, we don't have such atomic instruction. Thus, what we will do instead is to ask the hardware for a few useful instructions upon which we can build a general set of synchronization primitives.

The fundamental problem in concurrent programming

- We would like to execute a series of instructions atomically, but due to the conditions below, we can't:
    - The presence of interrupts on a single processor.
    - Multiple threads executing on multiple processors concurrently.

starvation

Does any thread contending for the lock starve while doing so, thus never obtaining it?

Fairness

Fairness

- Does each thread contending for the lock get a fair shot at acquiring it once it is freed?
- Does any thread contending for the lock starve while doing so, thus never obtaining it?

deadlock

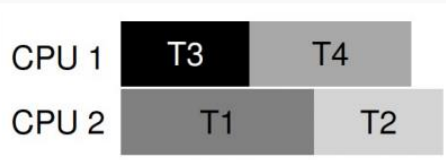
Deadlock

- When a producer and a consumer are each stuck waiting for each other.
- Four conditions need to hold for a deadlock to occur. If any of these four conditions are not met, deadlock cannot occur.
    - Mutual exclusion
    - Hold-and-wait
    - No pre-emption
    - Circular wait

Deadlock avoidance via scheduling

- Instead of deadlock prevention, in some scenarios deadlock avoidance is preferable.
- Avoidance requires some global knowledge of which locks various threads might grab during their execution, and subsequently schedule said threads in a way as to guarantee no deadlock can occur.
- Since such approaches can limit concurrency, it is not a widely used general-purpose solution.

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no



Detect and recover

- One final general strategy is to allow deadlocks to occasionally occur, and then take some action once such a deadlock has been detected.
- Many database systems employ deadlock detection and recovery techniques.

Non-deadlock bugs

- Non-deadlock bugs make up a majority of concurrency bugs according to Lu's study.
- They are often easy to fix.
- To major types of non-deadlock bugs (according for 97% of non-deadlock bugs):
    - Atomicity violation bugs: the desired serializability among multiple memory accesses is violated.
    - Order violation bugs: the desired order between two (groups of) memory accesses is flipped.

locks and condition variables

Locks

- Programmers annotate source code with locks, putting them around critical sections, and thus ensure that any such section executes as if it were a single atomic instruction.
- By putting a lock around a section of code, the programmer can guarantee that no more than a single thread can ever be active within that code.
- Thus, locks help transform the chaos that is triggered by the OS scheduler into a more controlled activity.
- A lock is just a variable.
- Multiple locks: Instead of one big lock that is used any time any critical section is accessed (a coarse-grained locking strategy), one will often protect different data and data structures with different locks, thus allowing more threads to be in locked code at once (a more fine-grained approach).

The concerns while building locks

- Mutual exclusion
- Fairness
- Performance

Condition variables

- A condition variable is an explicit queue that threads can put themselves on when some state of execution (i.e., some condition) is not as desired (by waiting on the condition).
- Some other thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue.

- A thread wishes to check whether a condition is true before continuing its execution. Condition variables can be used to serve this purpose.

semaphores in C

Semaphores

- One can use semaphores as both locks and condition variables.
- A semaphore is an object with an integer value that we can manipulate with two routines.
    - In the POSIX standard, these routines are sem_wait() and sem_post()
- Because the initial value of the semaphore determines its behavior, before calling any other routine to interact with the semaphore, we must first initialize it to some value.

Wait

- sem_wait() will either
    - return right away (because the value of the semaphore was one higher when we called sem_wait()), or
    - will cause the caller to suspend execution waiting for a subsequent post.
- Of course, multiple calling threads may call into sem_wait() and thus all be queued waiting to be waken.

Post

- sem_post() does not wait for some particular condition to hold like sem_wait() does.
- Rather, it simply increments the value of the semaphore and then, if there is a thread waiting to be wakened, wakes one of them.

The value of semaphore

- The value of the semaphore, when negative, is equal to the number of waiting threads.

Binary semaphores

- Locks only have two states (held and not held)
- We call a semaphore used as a lock a binary semaphore.
- The initial value of the binary semaphore should be 1.

Semaphores for ordering

- Semaphores can also be used as conditional variables.
- Semaphores are useful to order events in a concurrent program.

understand the assembly code in the chapters of "Introduction to Concurrency" and "Locks"

such as: (1) load memory location x into a register, (2) add 1 to that register, and (3) store the result to memory location x. If we disassemble

Thread A

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

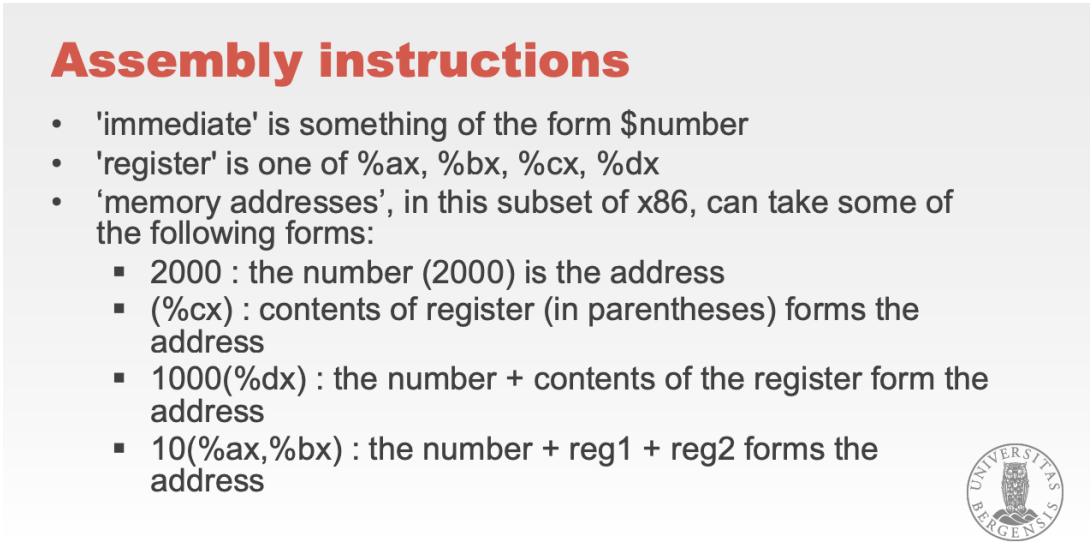
From weekly exercises week 6 task 1. Related to questions at the end of chapter “Concurrency: An introduction”.

```
mov immediate, register     # moves immediate value to register
mov memory, register        # loads from memory into register
mov register, register      # moves value from one register to other
mov register, memory        # stores register contents in memory
mov immediate, memory       # stores immediate value in memory

add immediate, register     # register  = register  + immediate
add register1, register2    # register2 = register2 + register1
sub immediate, register     # register  = register  - immediate
sub register1, register2    # register2 = register2 - register1

neg register                # negates contents of register

test immediate, register    # compare immediate and register (set condition codes)
test register, immediate    # same but register and immediate
test register, register     # same but register and register

jne                         # jump if test'd values are not equal
je                          #                       ... equal
jlt                         #     ... second is less than first
jlte                        #               ... less than or equal
jgt                         #            ... is greater than
jgte                        #               ... greater than or equal

push memory or register     # push value in memory or from reg onto stack
                            # stack is defined by sp register
pop [register]              # pop value off stack (into optional register)
call label                  # call function at label

xchg register, memory       # atomic exchange:
                            #   put value of register into memory
                            #   return old contents of memory into reg
                            # do both things atomically

yield                       # switch to the next thread in the runqueue

nop                         # no op
```

there can be tasks related to C and assembly code

Pointer

- The pointer in C language is a variable which stores the address of another variable.
- This variable can be of type int, char, array, function, or any other pointer.
- Pointers in C language are widely used in arrays, functions, and structures. It reduces the code and improves the performance.
- It makes it possible to access memory location in the computer's memory.



Persistence:

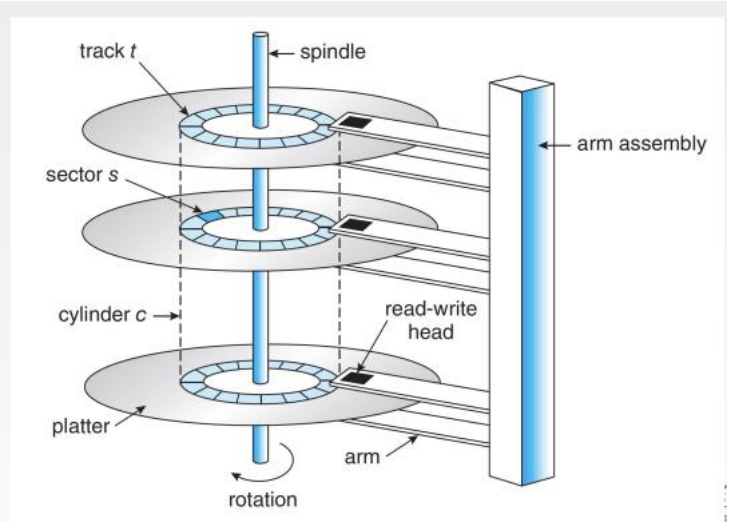
the terminology of different parts of the hard disks

The interface

- Address space of the drive
    - The drive consists of a large number of sectors (512-byte blocks)
    - The sectors are numbered from 0 to n-1 on a disk with n sectors
    - 0 to n-1 is thus the address space of the drive
- Multi-sector operations
    - Many file systems read and write 4KB at a time (or more)
    - A single 512-byte write on disk is atomic. It will either complete in its entirety or it won't complete at all.
- Assumptions
    - Accessing two blocks near one-another within the drive's address space will be faster than accessing two blocks that are far apart.
    - Accessing blocks in a contiguous chunk is usually much faster than any more random-access pattern.

Basic geometry

- Platter
    - A circular hard surface on which data is stored persistently by inducing magnetic changes to it.
    - A disk may have one or more platters.
    - Each platter has 2 sides, each of which is called a surface.
- Spindle
    - Spindle is connected to a motor that spins the platters around at a constant rate.
- Track
    - Data is encoded on each surface in concentric circles of sectors.
    - A concentric circle is called a track.
    - A single surface contains many thousands and thousands of tracs.
- Disk head
    - One disk head per surface of the drive.
    - The process of reading and writing from the surface is accomplished by the disk head.
- Disk arm
    - The disk head is attached to a single disk arm.
    - A disk arm moves across the surface to position the head over the desired track.



rotational delay

Single-track latency: the rotational delay

- To read a specific sector S, it must wait for S to rotate under the disk head.
- This delay is part of the I/O service time.

seek time

Multiple tracks: seek time

- The complete I/O time includes:
    - Seek time: the time spent to move the arm to the correct track
        - Acceleration phase
        - Coasting phase
        - Deceleration phase
        - Settling phase
    - Rotational delay
    - Data transfer time
- Note that while the disk arm is moving to the correct track, the platter is rotating, too.

transfer time

The amount of time it takes to transfer data (when someone issues an I/O).

disk scheduling (shortest seek time first (SSTF), shortest positioning time first (SPTF))

Shortest seek time first (SSTF)

- Picking requests on the nearest track to complete first.
- Problems of SSTF:
    - Drive geometry is not available to the host OS, rather, it sees an array of blocks
    - Solution: nearest-block-first (NBF)
    - Starvation
        - Solution: SCAN

Shortest position time first (SPTF)

- What it depends on is the relative time of seeking as compared to rotation.
    - If seek time is much higher than rotation delay, then SSTF are just fine.
    - If seek is quite a bit faster than rotation, then it would make more sense to seek further to service requests, then to perform a shorter seek where the sector has to rotate all the way around before passing under the disk.

the pros and cons of using different RAIDs

Redundant arrays of inexpensive disks (RAID)

- The term was introduced in the late 1980s by a group of researchers at U.C. Berkley.
- They wanted to use multiple disks to build a better storage system.
- RAID is like a computer system.
    - It runs specialized software designed to operate a group of disks.
    - It consists of multiple disks, memory, and one or more processors.
- RAIDS transform a number of independent disks into a large, more capacious, and more reliable single entity.

Advantages of RAIDs

- RAIDs offer a number of advantages over a single disk:

    - Capacity
    - Reliability
    - Performance
- Transparency enables deployment.
    - It enables one to simply replace a disk with a RAID and not change a single line of software.
- RAIDs are designed to detect and recover from certain kinds of disk faults. Thus, knowing exactly which faults to expect is critical in arriving upon a working design.

RAID levels

- RAID level 0 (block striping)
    - Serves as an excellent upper-bound on performance and capacity.
    - This approach is designed to extract the most parallelism from the array when requests are made for contiguous chunks if the array.
    - Reliability: any disk failure will lead to data loss.

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

- RAID level 1 (mirroring)
    - We make more than one copy of each block in the system. Each copy should be placed on a separate disk. By doing so we can tolerate disk failure.
    - Capacity: half of our peak useful capacity.
    - Reliability: it can tolerate the failure of any one disk.
    - Performance: not as food as with the RAID level 0.

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

- RAID-4 (block-striping plus a single parity disk)
    - Parity-based approaches overcome the huge space penalty paid by mirrored systems. However, they do so at a cost: performance
    - Reliability: RAID-4 tolerates 1 disk failure and no more. If more than one disk is lost, there is simply no way to reconstruct the lost data.
    - The parity disk is a bottleneck under certain types of workload.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

- RAID-5 (block-striping with rotating parity)
    - Works almost identically to RAID-4, except that it rotates the parity block across drives.
    - It removes the parity-disk bottleneck for RAID-4.
    - Similar to RAID-4 when it comes to capacity, reliability and performance. The only exception is with random read and write.
    - Because RAID-5 is basically identical to RAID-4 except in the few cases where it is better, it has almost completely replaced RAID-4 in the marketplace.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

RAID comparison: a summary:

	RAID-0	RAID-1	RAID-4	RAID-5
Capacity	$N \cdot B$	$(N \cdot B)/2$	$(N-1) \cdot B$	$(N-1) \cdot B$
Reliability	0	1 (for sure) $\frac{N}{2}$ (if lucky)	1	1
Throughput				
Sequential Read	$N \cdot S$	$(N/2) \cdot S^1$	$(N-1) \cdot S$	$(N-1) \cdot S$
Sequential Write	$N \cdot S$	$(N/2) \cdot S^1$	$(N-1) \cdot S$	$(N-1) \cdot S$
Random Read	$N \cdot R$	$N \cdot R$	$(N-1) \cdot R$	$N \cdot R$
Random Write	$N \cdot R$	$(N/2) \cdot R$	$\frac{1}{2} \cdot R$	$\frac{N}{4}R$
Latency				
Read	T	T	T	T
Write	T	T	$2T$	$2T$

the tolerance of disk failure

RAID level 0

- Reliability: any disk failure will lead to data loss.

RAID level 1

- Reliability: it can tolerate the failure of any one disk.

RAID-4

- Reliability: RAID-4 tolerates 1 disk failure and no more. If more than one disk is lost, there is simply no way to reconstruct the lost data.

RAID-5

- Similar to RAID-4 when it comes to reliability.

the recovery of missing data

hard links, symbolic links

Hard links

- The command ln "links" a new file name to an old one. Essentially, it creates another way to refer to the same file.
- link() creates another name in the directory you are creating the link to, and refers it to the same inode number (i.e., low-level name) of the original file.
- The file is not copied in any way; rather, you now just have two readable names (file and file2) that both refer to the same file.
- Hard links are somewhat limited:
    - You can't hard link to files in other disk partitions.
        - Because inode numbers are only unique within a particular file system, not across file systems, etc.
    - Thus, symbolic links was created.

Symbolic links

- A symbolic link is also called a soft link.

- To create such a link: ln and the -s flag.
- A symbolic link is actually a file itself, of a different type.
- Unlike hard links, removing the original file named file causes the link to point to a pathname that no longer exists. file2 becomes the dangling reference.

The permission bits

Permission bits

- To see permissions for a file foo.txt, type ls -l
- The first bit:
    - Files (-), directories (d) and symbolic links (l)
- The permissions consist of three groupings:
    - Owner
    - Someone in a group
    - Anyone else
- The abilities of the owner, group members, and others include the ability to
    - Read
    - Write
    - Execute
- To change the permissions: chmod 600 foo.txt
- The execute bit for directories: it enables a user (or group, or everyone) to do things like change directories (i.e., cd) into the given directory, and, in combination with the writable bit, create files therein.

very simple file system (vsfs)

Very simple file system (vsfs)

- This file system is a simplified version of a typical UNIX file system and thus serves to introduce some of the basic on-disk structures, access methods, and various policies that you will find in many file systems today.
- We use vsfs to introduce most concepts of file systems.

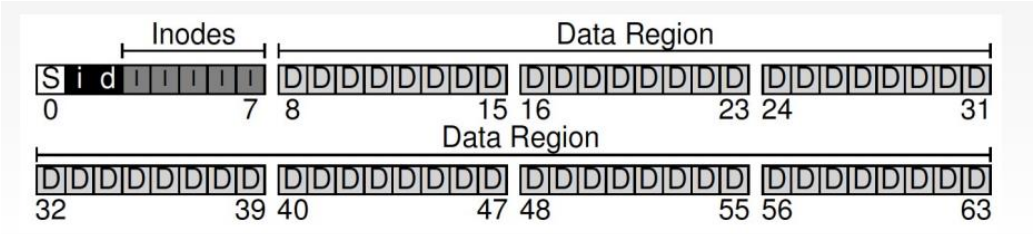
Data structures of the file system

- vsfs uses arrays of blocks or other objects.

- More sophisticated file systems, like SGI's and XFS, user more complicated tree-based structures.
- The view of the disk partition where we're building our file system is:
    - A series of blocks, each of size 4 KB
    - The blocks are addressed from 0 to N-1, in a partition of size N 4-KB blocks.

What we need to store in these blocks in order to build a files system

- Data region: most of the space in any file system is (and should be) user data. This region is called the data region.
- Index node (inode)
    - The metadata of a file is stored in the inode of the file.
- Inode table
    - The inode table holds an array of on-disk inodes.
- A bitmap
    - Each bit in the bitmap is used to indicate whether the corresponding inode/data block is free (0) or in-use (1).
- Superblock
    - The superblock contains information about this particular file system. For example:
        - Number of inodes and data blocks in the system.
        - Where the inode table begins.
        - The file system type, e.g., vsfs.



Directories are just a specific type of file that store:

- (name, inode_number) pairs.

FFS

Fast File System (FFS)

- The idea was to design the file system structures and allocation policies to be “disk aware” and thus improve performance.
- FFS divides the disk into a number of cylinder groups. Here is a depiction of what FFS keeps within a single cylinder group:

- The basic mantra is simple:
  - Keep related stuff together
  - Keep unrelated stuff far apart

The placement of directories

- FFS finds the cylinder group with
  - a low number of allocated directories (to balance directories across groups), and
  - a high number of free inodes (to subsequently be able to allocate a bunch of files)
- and puts the directory data and inode in that group.

The placement of files

- Principle 1
  - FFS allocates the data blocks of a file in the same group as its inode, thus preventing long seeks between inode and data.
- Principle 2
  - FFS places all files that are in the same directory in the same cylinder group of the directory they are in.

The large file exception

- Without a different rule, a large file would entirely fill the block group it is first placed within (and maybe others). This is undesirable since it may hurt file-access locality.
- Solution:
  - With the large file exception, FFS instead spreads the file across groups, and the resulting utilization within any one group is not too high.

- However, spreading blocks of a file across the disk will hurt performance, particularly in the relatively common case of sequential file access.
- We can address this problem by choosing chunk size carefully.

crash-consistency problem

The crash-consistency problem

- What we'd like to do ideally is move the file system from one consistent state (e.g., before the file got appended to) to another atomically (e.g., after the inode, bitmap, and new data block have been written to disk).
- Unfortunately, we can't do this easily because the disk only commits one write at a time, and crashes or power loss may occur between these updates.

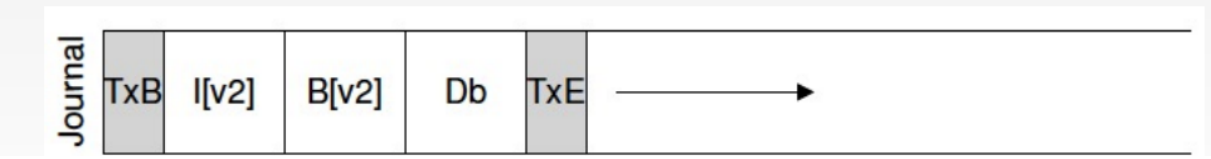
journaling and recovery

Journaling (or write-ahead logging)

- When updating the disk, before overwriting the structures in place, first write down a little note (somewhere else on the disk, in a well-known location) describing what you are about to do.
- By writing the note to disk, you are guaranteeing that if a crash takes place during the update (overwrite) of the structures you are updating, you can go back and look at the note you made and try again.
- Thus, you will know exactly what to fix (and how to fix it) after a crash.

Data journaling

- We wish to write the inode, bitmap, and data block to disk.
- Before writing them to their final disk locations, we write them to the log.



- The file system issues the transactional write in steps.
    - First, it writes all the blocks except the TxE block to the journal, issuing these writes all at once.

    - When those writes complete, the file system issues the write of the TxE block.

Recovery

- If the crash happens before the transaction is written safely to the log, then the pending update is simply skipped.
- If the crash happens after the transaction has committed to the log, but before the checkpoint is complete, the file system can recover the update.
    - Redo logging: When the system boots, the file systems recovery process will scan the log and replay the transaction.
    - Luckily, system crashes do not happen so often.

Reuse the log space

- Journaling file systems treat the log as a circular data structure, re-using it over and over (therefore circular log).
- Once a transaction has been checkpointed, the file system should free the space it was occupying within the journal, allowing the log space to be reused.

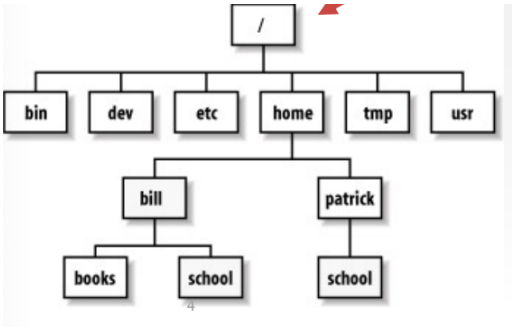
The protocol to update the file system

1. Journal write
2. Journal commit
3. Checkpoint
4. Free

Linux file system

Linux directory structure

- In Linux, all files and directories are organized in a tree structure.
- Each user only has write access to their own home folder and must obtain elevated permissions (become the root user) to modify other files on the system.



there can be tasks related to Linux command lines and shell scripting