Course Summary

ID1206 OPERATING SYSTEMS

Structure of the Exam

- The question are from the book, lectures, seminars and assignments
- 5 parts
  - Processes
  - Communication (concurrency, sockets etc),
  - Scheduling
  - Virtual memory
  - File Systems and Storage

How to Study for the Exam?

- Read the book
- Go through the lectures
- Do the assignments
- You are allowed to have an A4 handwritten paper with you

UNIX Commands

ls – list files and directories

mkdir – create directory

rmdir – remove directory

cd – change directory

pwd – path to current directory

touch – create a file

rm – remove files or directories

mv – move a file

cp – copy files or directories

ln – create a link to a file

chmod – change permissions of a file

cat – print file to standard output

echo – display a line of text

head – output the first part of file

tail – output the last part of file

diff – compare files line by line

sort – sort lines of text files

wc – newline, word and byte count for file

sed – stream editor

grep – find a pattern in a file

tr – translate or delete characters

man – manual

Unix Commands – Exam Question

If we give the following commands after each other in a *shell*; what will the result be?

```
> mkdir foo
> cd foo
> echo "hello hello" > tomat.txt
> mkdir ../bar
> ln tomat.txt ../bar/gurka.txt
> rm tomat.txt
> cd ../
> wc -w bar/gurka.txt
```

Create directory “foo”

Write “hello hello” to tomat.txt

Create directory “bar” outside directory “foo”

Create hard link from tomat.txt (in directory “foo”) to gurka.txt (in directory “bar”)

Remove tomat.txt

Count the words in file gurka.txt

Answer: 2

Processes

- In which segments data is stored
  - Global – global variables, variables outside functions
  - Heap – data allocated using malloc()
  - Stack – local variables, function arguments, return values
- How processes are created
  - fork()

Processes - Exam Question

- Answer:

  - x is global

  - h[], i and c are on the stack

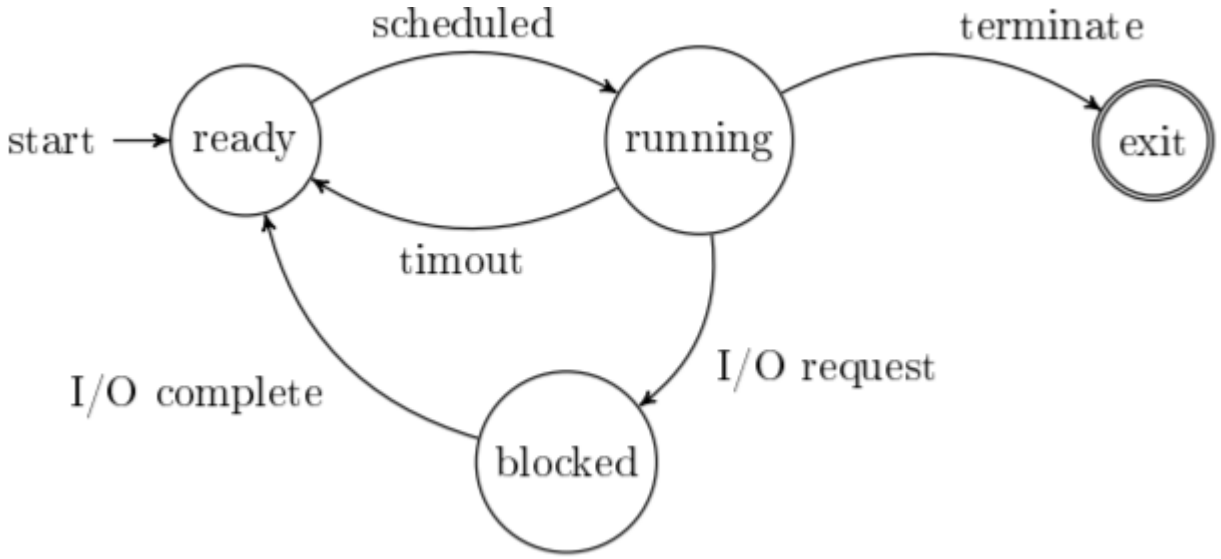
  - Printout is h[3] = 4

In the code below we have the variables: `x`, `i`, `h` och `c`. In which segments do we find the data stuctuctures they are assigned to: global, stack or heap? What will be printed on stdout?

```
#include <stdio.h>
#include <stdlib.h>

int x = 3;

int foo(int i) {
  int h[] = {1,2,3,4};
  return h[i];
}

int main() {

  int c = foo(x);
  printf("h[%d] = %d \n", x, c);
  return 0;
}
```

Scheduling

- How are processes scheduled
  - In what order should they run?
  - Different states of processes



Scheduling – Turnaround and Response Time

- Turnaround time
  - Turnaround time = Finish time – Arrival time
  - The time it takes for a process to finish execution minus the time when the process entered the system
- Response time
  - Response time = First run time – Arrival time
  - The time a process first gets to run minus the time the process entered the system

Scheduling - Algorithms

- First in first out (FIFO)
  - The processes run to completion in the order in which they came in
- Shortest job first (SJF)
  - Runs the processes that has the shortest execution time to run first
- Shortest time to completion first (STCF)
  - Like SJF but it is possible to stop and run processes alternately
- Round Robin (RR)
  - Runs processes in time slices
  - Good for response time but bad for turnaround time

Scheduling - Multilevel feedback queue (MLFQ)

- Problems with algorithms such as SJF and STCF is that we cannot know in advance how long a program will run -> multilevel feedback queue addresses this problem
- It decides which processes run first by observing their behavior and assigning different priorities. Processes with high priority are chosen to run first.
  - If a process uses the CPU under a long amount of time -> low priority
  - If a process gives away CPU time often -> 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 RR.
  - **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.

Scheduling - Exam Question

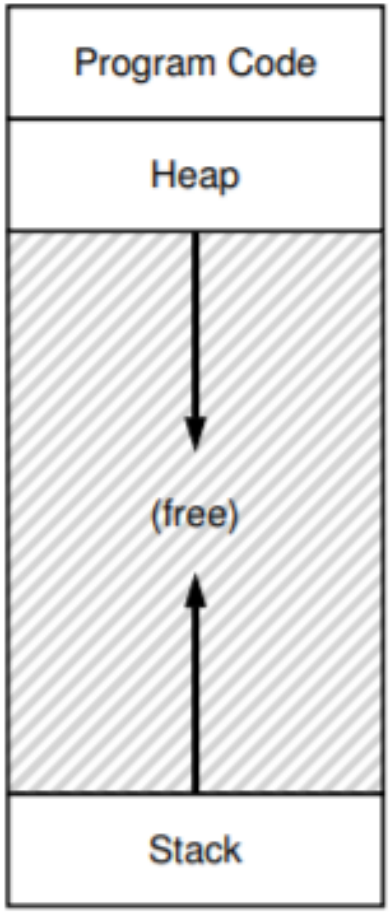
Assume we have a scheduler that implements "shortest job first" i.e. not able to preempt jobs. If we have three jobs that will take 10ms, 20ms and 30ms it's a better strategy than taking the jobs in random order, show why.

Answer:

- Turnaround time = (10 + 30 + 60) / 3 = 33 ms
- Wrong order could give turnaround time = (30 + 50 + 60) / 3 = 47 ms

Virtual Memory

- Address space
  - A running program's view of the memory
  - Virtual addresses are translated to physical addresses
- Ways of managing and dividing up the memory
  - Paging, segmentation, base and bounds



Virtual Memory – Base and Bounds Registers

- Each process has one base and bounds pair
- Base register – where in the physical memory the address space begins
- Bounds register – where the address space ends, size of address space, maximum amount of memory available to a process
- Physical address = virtual address + base
- Problem: By having a single base and bound pair means that the code, stack and heap are all in the same memory region. The address space has a lot of unused memory between the heap and the stack and this leads to internal fragmentation

Virtual Memory - Segmentation

- Divides memory into variable sized blocks called segments
- One base and bound pair per segment of the address space (code, stack and heap)
- The segments can be placed individually in physical memory
- Problem: since the segments are of variable sizes the memory gets divided into different sizes of non-contiguous pieces which leads to external fragmentation

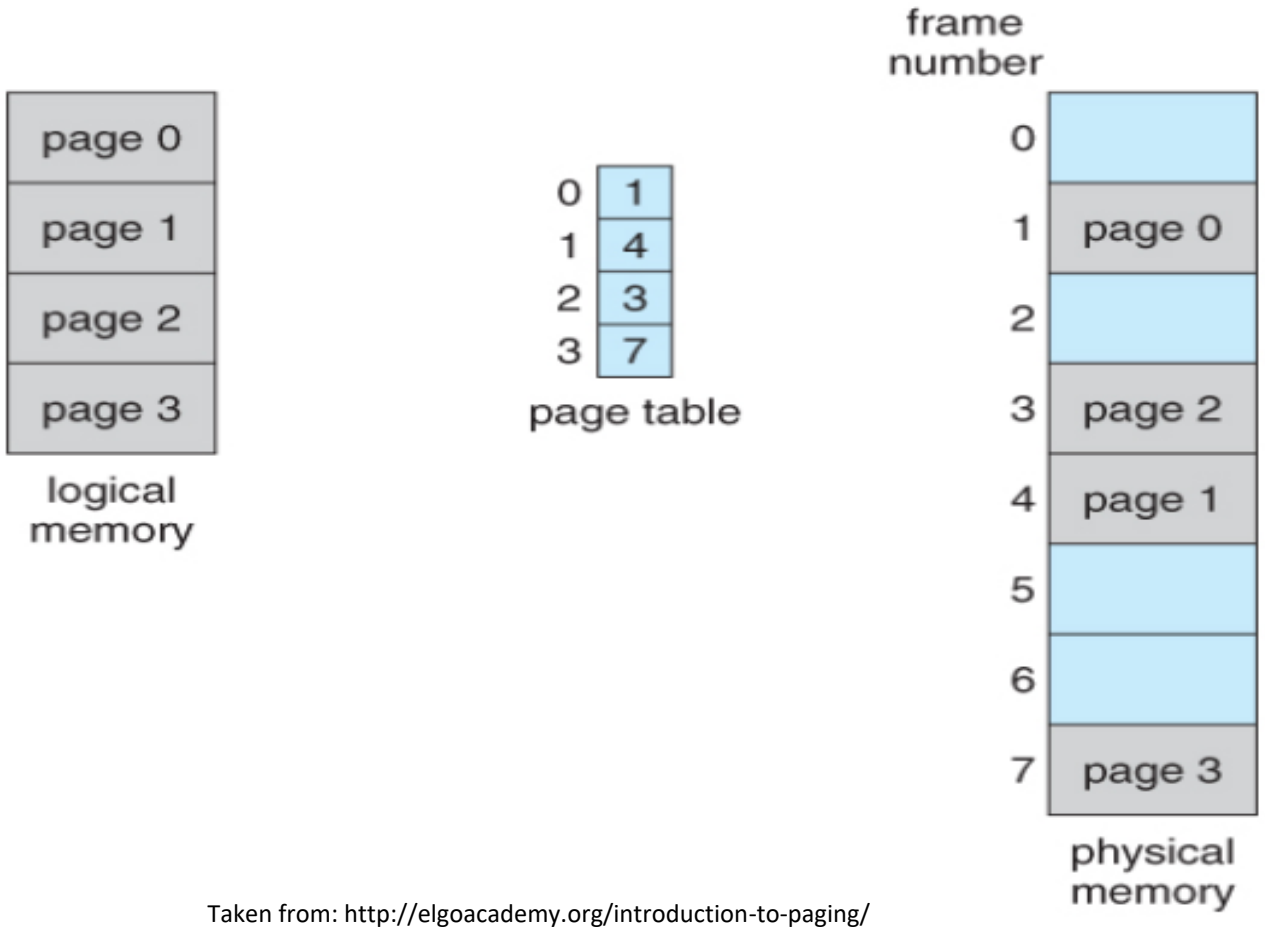
Virtual Memory - Paging

- Divides physical memory into equal sized blocks -> called frames
- Divides virtual address space into blocks of the same size -> called pages
- Each page corresponds to a frame

Virtual Memory - Page table

- Data structure that specifies which virtual page points to which physical frame
- One per process, contains one entry for each page of the process
- Used to translate virtual addresses to physical addresses
- Takes a virtual page number and returns a physical frame number
- Can have different structures: linear, hierarchical (multilevel)

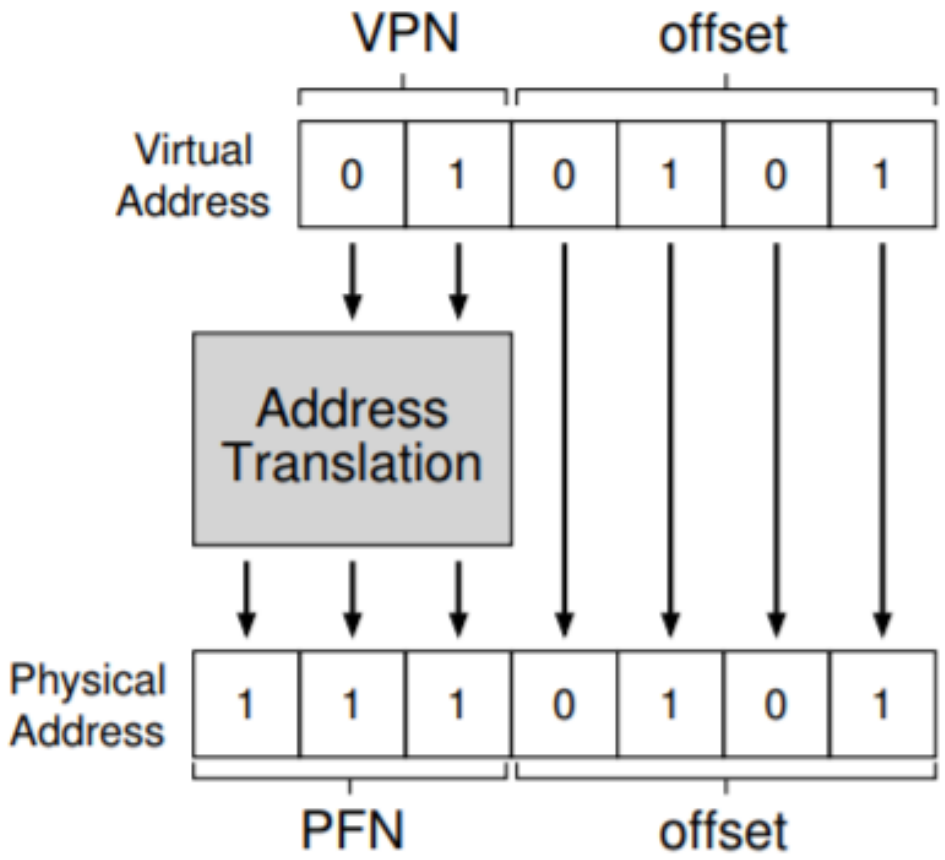
Virtual Memory - Page table example



Taken from: http://elgoacademy.org/introduction-to-paging/

Virtual Memory – Paging Address Translation

- Translate virtual address 010101
  - VPN: virtual page number
  - offset: tells us which byte of the page we want
- Get the page frame number (PFN) from page table
- Replace PFN with VPN
- Offset stays the same
- Corresponding physical address: 1110101

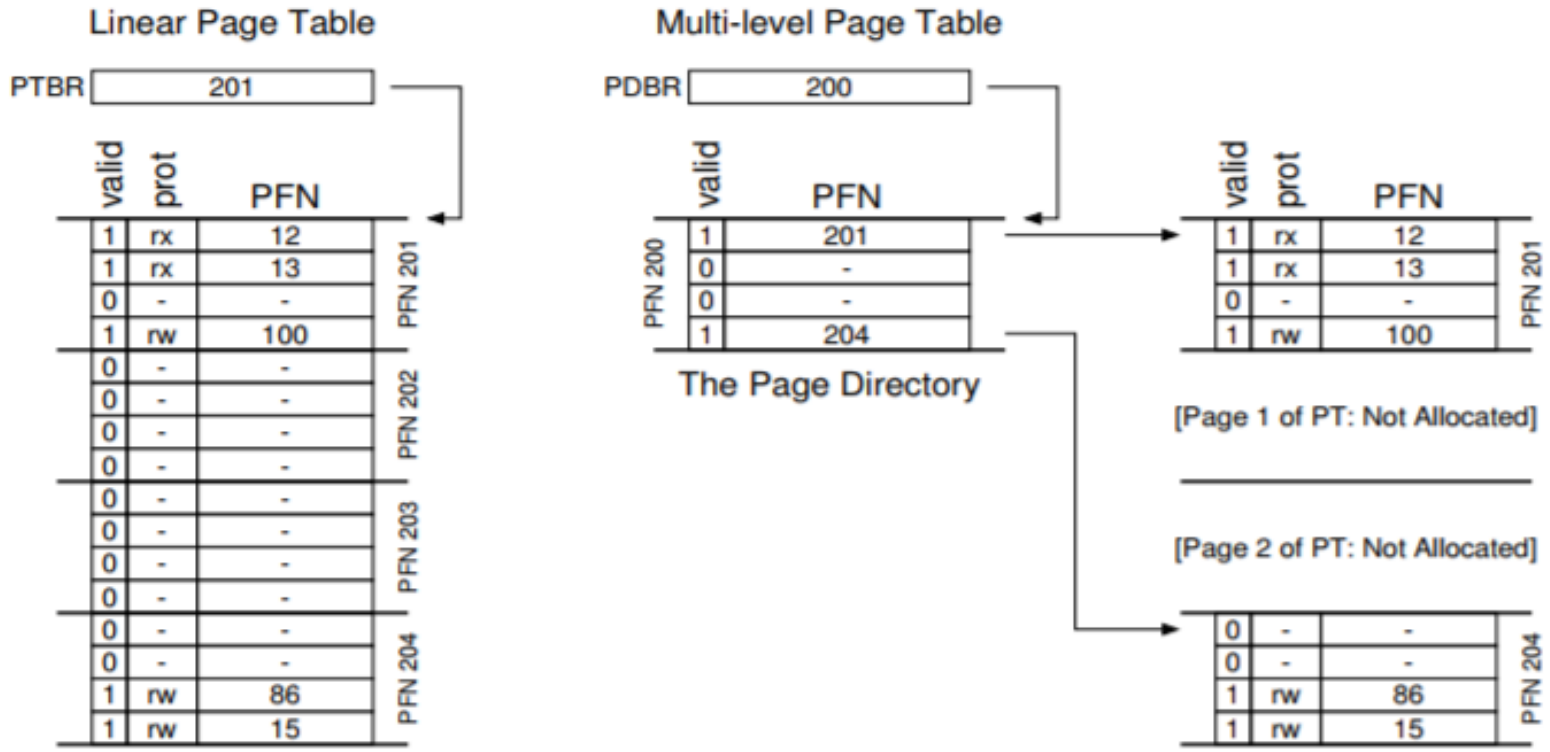


Taken from: Operating Systems: Three Easy Pieces by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau

Virtual Memory – Multilevel Page Table

- Problems with linear page tables is that they may become very large and allocate unused memory
- Multilevel page tables are the solution to these problems
- The linear page table is divided into equal sized smaller pieces, also called pages
- One page contains several entries of the page table
- The page directory holds these entries
- The entries in the page directory then point to the page in the page table
- The page in the page table then points to the physical frame

Virtual Memory – Multilevel Page Table: example



Taken from: Operating Systems: Three Easy Pieces by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau

Virtual Memory - Swapping

- What if we run out of physical memory? There is not room for all pages
- Swap space – Space on the disk to move pages in to and out from physical memory
- Take frames from physical memory and swap them (write them) to disk
- If a swapped out page is accessed we swap them back into memory
- Page fault: accessing a page that has been swapped out of physical memory

Virtual Memory – Swapping Policies

- What pages should be swapped out?
- There exist different policies to decide which page we should evict
  - **First in first out (FIFO):** pages that came in to memory first are swapped out first
  - **Random:** chooses a page randomly
  - **Least recently used (LRU):** swaps out the page that was used least recently, if a program has accessed a page recently it will likely access it again soon
  - **Clock algorithm:** approximation of LRU, each page has a use bit of 0 or 1.
    If 1 –> page has recently been accessed, should not be swapped out
    If 0 –> page can be swapped out

Virtual Memory - Exam Question

Assume we have a virtual address that consist of a 22-bit page number and a 12-bit offset. We have a physical address space of 32 bits, encode a frame number in 20 bits and elements in a page table are 4 bytes. How large is a page and how large would a complete page table be?

Answer:

12 bit offset => 2^12 page size = 4KiB page size.

22 bit page number (VPN) => 2^22 pages = 4Mi pages. One entry in the page table is 4 bytes => 4Mi*4 byte entries = 16MiB page table

Memory Management

- How memory is managed ;)
- There exist different ways of dividing up memory
- The two main ways to do it are segmentation and paging

Two main problems:

- External fragmentation (segmentation)
  - Pieces of memory are spread out and are to small for a process to use
- Internal fragmentation (paging)
  - A process gets allocated more memory than it needs and some memory goes unused

Memory Management – Free List

- Data structure used for dynamic memory allocation
- Contains blocks of the available free space on the heap

Memory Management – Splitting and Coalescing

- Common techniques used in memory allocation
- Split a memory block in the free list
  - Return the needed amount to the user
  - Keep the rest
- When coalescing you merge two blocks of free memory that lies next to each other
  - Good to make sure that the heap is not divided into several small blocks of memory

Memory Management – Allocation Strategies

- There exist different strategies used to select the appropiate memory that is requested from the user
- Best fit
- Worst fit
- First fit
- Next fit
- Buddy allocation
- Slab allocator

Memory Management - Best Fit

- Searches through the free list and finds a memory block that is equal or larger than the requested memory
- Returns the requested amount to the user and keeps the rest
- Pros – Reduces the amount of wasted memory
- Cons – A lot of performance is required to search through the whole list

Memory Management - Worst Fit

- Searches through the free list and finds the largest amount of free memory
- Returns the requested amount to the user and keeps the rest
- Pros – Reduces the amount of small memory, leaves big pieces of memory free
- Cons – A lot of performance, the whole list needs to be searched through, takes away large chunks of memory that could be needed

Memory Management - First Fit

- Finds the first block of memory that is large enough
- Returns the requested amount to the user and keeps the rest
- Pros – Quick, no need to search through the whole list
- Cons – Could leave small blocks of memory in the beginning of the list

Memory Management - Next Fit

- Similar to first fit but has an extra pointer that keeps track of where you were last time and begins the search from there next time
- Performance is similar to first fit

Free Memory Strategies - Example

Which block will each strategy choose for a request of size 15?

- Best fit: 20
- Worst fit: 30
- First/Next fit: 30

Taken from: Operating Systems: Three Easy Pieces by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau

Memory Management – Segregated Lists

- If a particular application has one (or a few) popular-sized requests that it makes, keep a separate list just to manage objects of that size
- In this approach fragmentation is much less of a concern; moreover, allocation and free requests can be served quite quickly when they are of the right size, as no complicated search of a list is required
- How much memory should one dedicate to the pool of memory that serves specialized requests of a given size, as opposed to the general pool?

Memory Management – Slab Allocator

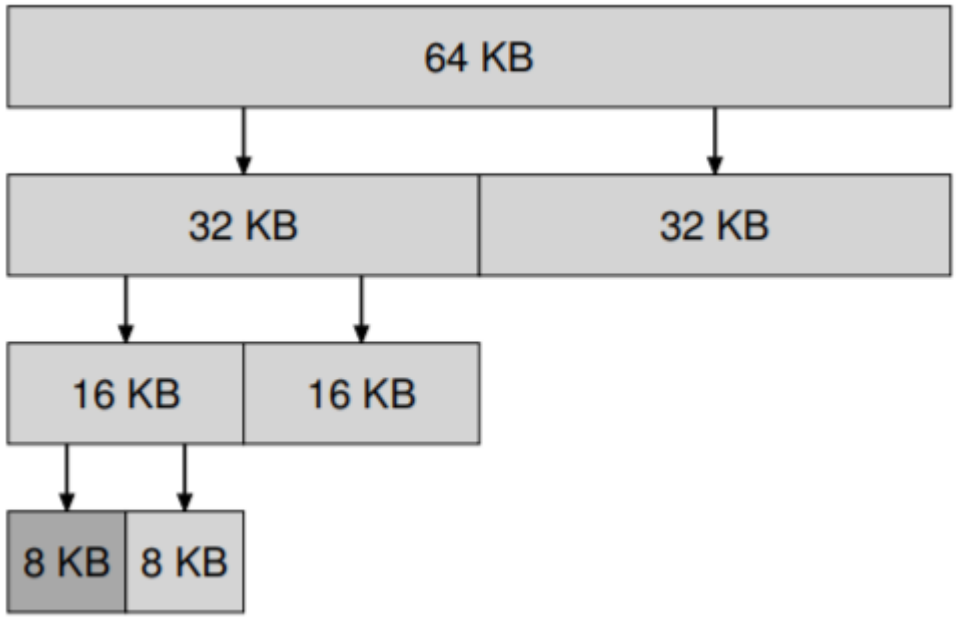
- Handles the above issue well
- When the kernel boots up, it allocates a number of object caches for kernel objects that are likely to be requested frequently (e.g. locks, inodes etc)
- The object caches thus are each segregated free lists of a given size and serve memory allocation and free requests quickly
- When a given cache is running low on free space, it requests some **slabs** of memory from a more general memory allocator
- The memory allocator can in turn request memory from the specialized source (the segregated list) if it is not used much
- The slab allocator also initializes free objects on the list beforehand, making the process even faster

Memory Management – Buddy Allocation

- The free memory is seen as 2^n
- When a request for memory is made the memory is recursively splitted in two until an appropiate size is found
- Pros – Easy to determine if two blocks can be merged together
- Cons – Internal fragmentation as only blocks of size 2^n can be given

Memory Management - Buddy Allocation

- 64 KB free space
- User request for 7 KB block of memory
- The memory block is divided in 2^n
- 8 KB is returned to the user



Taken from: Operating Systems: Three Easy Pieces by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau

Memory Management - Exam Question

One strategy to find a suitable memory block is to find the block that best suites our needs (without being too small); this must by all aspects be the a good strategy. Another approach is to simply take the first block that is found even if it is considerably larger than what we need. What would the benefit be for the latter strategy and what is the possible downside?

Answer: This approach is called “first fit”. The upside is that it is quick, we don’t have to go through the list to find a free block (as supposed to “best fit” that is described in the first stratergy in the question). The downside is that this can lead to a lot of very small blocks in the beginning of the list, so the bigger block of memory that is requested, the longer it will take to find a suitable free block as we will have to search further into the list for bigger blocks.

Concurrent Programming

- Parallelize a program with the help of threads
- Speed up the execution by having several threads perform task concurrently

Concurrent Programming – Threads & Context switch

- Threads
  - Similar to a process but they share the same address space, except for the stack which is one per thread
  - A process can have several threads that has their own task to parallelize the program
  - Since threads share their data problems may occur for example when two threads update the same global variable due to context switches
- Context switch
  - When the CPU switches to run another process or thread
  - The state of the old process is saved

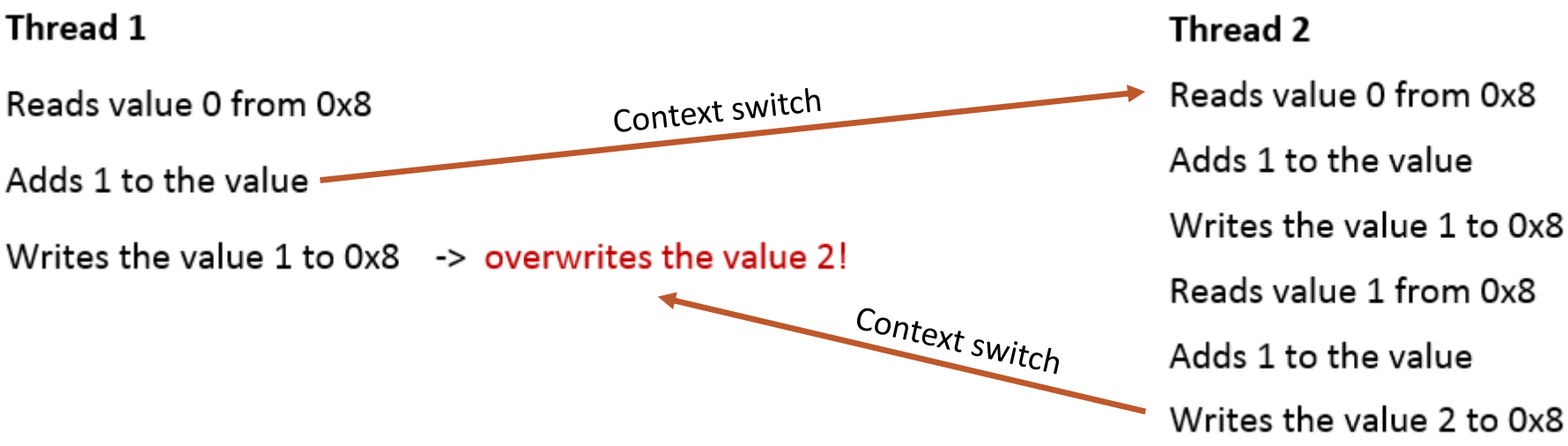
Concurrent Programming – Counter Example

- Two threads adding a global counter, in C code we have:

```
int count = 0;
count = count + 1;
```

- The assembly code when the c code is compiled is actually:

```
mov 0x8, %eax        //read value from 0x8 and move to %eax
add $1, %eax         //add 1 to %eax
mov %eax, 0x8        //store back value from %eax to 0x8
```



Concurrent Programming – Critical Section

- Section that accesses a shared resource such as a variable or data structure
- A piece of code that can't be executed by more than one thread concurrently
  - E.g.: `count = count + 1;`
- When several threads enters a critical section at the same time we get a **race condition** and the wrong result
- Mutual exclusion guarantees that race conditions do not occur
  - If one thread is executing in a critical section, no other threads can enter
  - Can be implemented by using locks

Concurrent Programming – Locks

- A kind of variable that is declared around the critical section
- The variable holds the state which is either “locked” or “unlocked”
- In the unlocked state no thread holds the lock
- In the locked state the lock is held by a thread and is in the critical section -> no other thread can enter
- The thread that holds the lock must call unlock() to allow other threads to acquire the lock

```
lock(&mutex);
count = count + 1;
unlock(&mutex);
```

Concurrent Programming – Problems

- Deadlock (Me first)
  - Processes are unable to proceed since they are all waiting on each other to release the locks
- Livelock (You first)
  - Processes are constantly changing states but no progress is achieved, threads yield at the same time and then also tries to take a lock at the same time -> no thread then acquires the lock
- Starvation (Some first, others never)
  - A process is never chosen to run due to greedy threads that never releases their locks

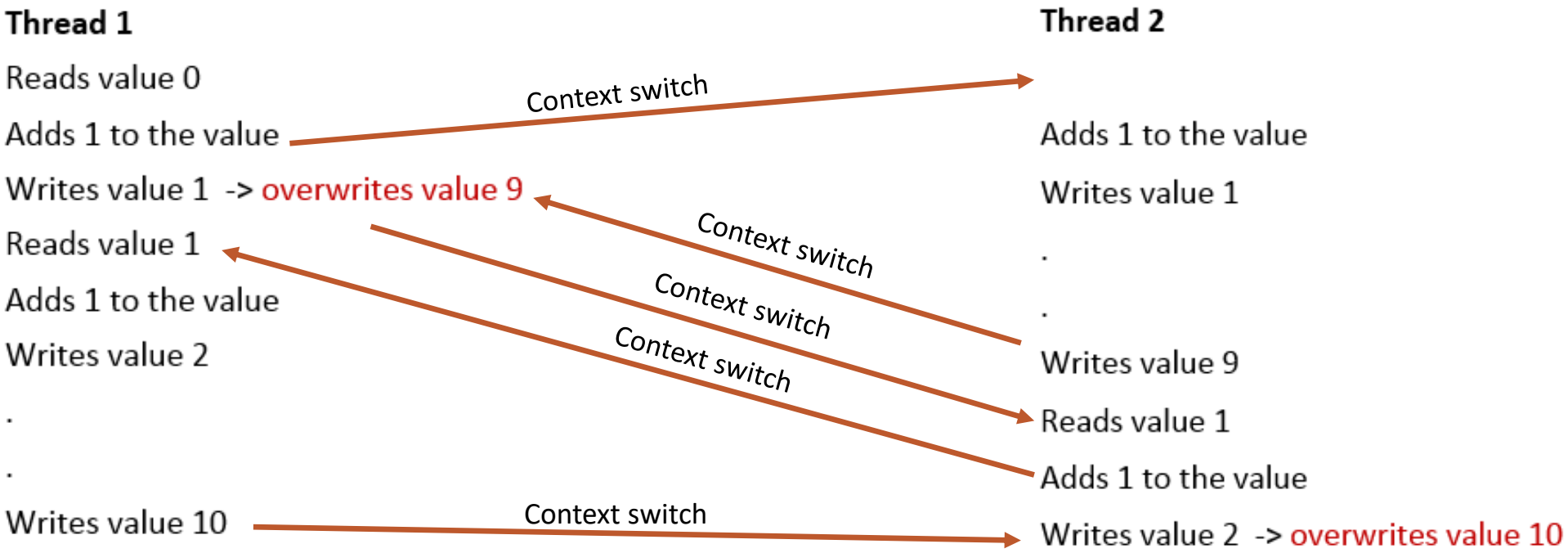
Concurrent Programming - Exam Question

If we execute the procedure `hello()` below in two threads concurrently, the result will be that `count` obtains the value ...- which values can `count` hold after both of the threads have completed the execution? Why is this possible?

```
int loop = 10;
int count = 0;

void *hello() {
  for(int i = 0; i < loop; i++) {
      count++;
  }
}
```

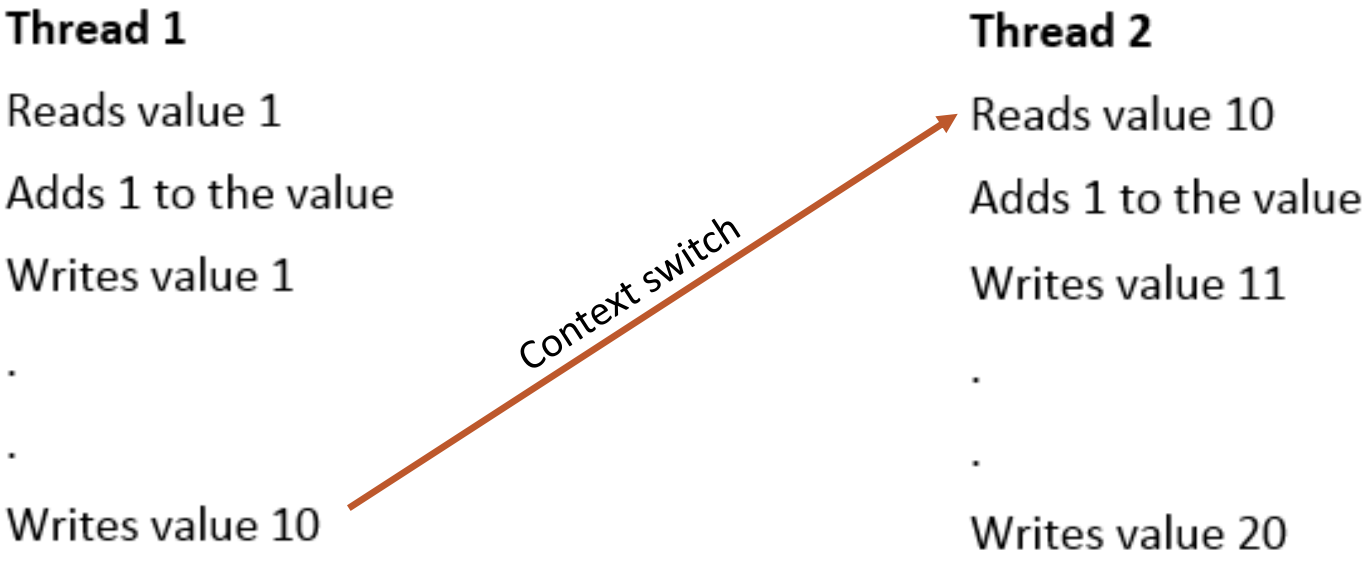
Answer: range is [2, 20]

Concurrent Programming – Exam Question Explanation



Answer is: 2

Concurrent Programming – Exam Question Explanation



Answer is: 20

Storage

This section is about how data is stored and accessed on different devices

- We need a persistent storage to save data that needs to survive a shutdown of the computer
- This can for example be an HDD or SSD

This part also includes interaction with different devices, which is the first part of this section

Storage – I/O devices

A device has two important components:

1. A hardware interface that it presents to the rest of the system
   - The interface allows the system structure to control the hardware operations
2. The internal structure
   - Implements the abstraction that the unit presents to the software

Storage - Buses

Buses are systems for communication between different components in the computer by transferring data between these components. There are different kinds of buses, for example:

1. Memory bus
   - Connects CPU to memory
2. I/O bus
   - Connects some devices to the system
3. Peripheral bus
   - Connects slower devices (mouse, discs etc) to the system.
   - For example USB

Devices that need higher performance are closer to the CPU (shorter bus => shorter access time)

Storage – Registers

A device can have different registers to allow the OS to control it:

- Status register: contains the current status of the unit
- Command register: tells the device to perform a certain operation
- Data register: used to pass data to the device or get data from the device

By reading from or writing to these registers the OS can control their behavior

Storage – Interaction Between OS and the Device

The protocol for interaction between the OS and a device follows the steps below:

1. The OS repeatedly polls a device (reads the status register) to find out if the device is ready to receive a command
2. The OS sends data to the device via the data register
3. The OS writes a command to the command register, which informs the device that data is ready available in the data register and that it should start executing the command
4. The OS waits for the device to complete the command

Storage – Interrupts

Instead of all that waiting, the OS can use interrupts and by this perform a context switch to do something else while waiting for the device

- The OS issues a request to the device, puts the calling process to sleep and context switch to another task
- When the device is done with the command, an interrupt is sent which starts an interrupt handler.
- The handler is a piece of OS code that will finish the request (for example read the data from the device) and wake up the waiting process

Whether interrupts or polling is more efficient depends on how fast the device is (context switching takes time too)

Storage - DMA

If the OS were to perform all transfers between devices this would take up too much of the OS's time. Solution:

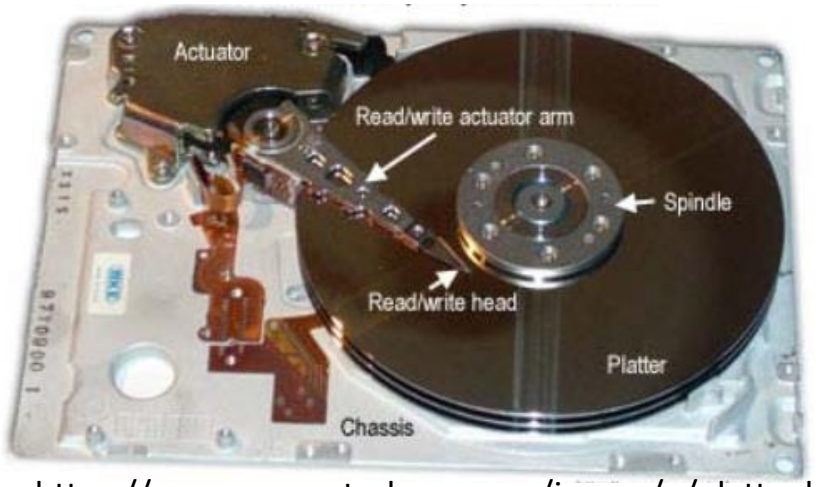
- Direct Memory Access (DMA)
  - A device that transfers data between devices and memory without involving the OS (that much)

Storage – The Hard Disk Drive (HDD)

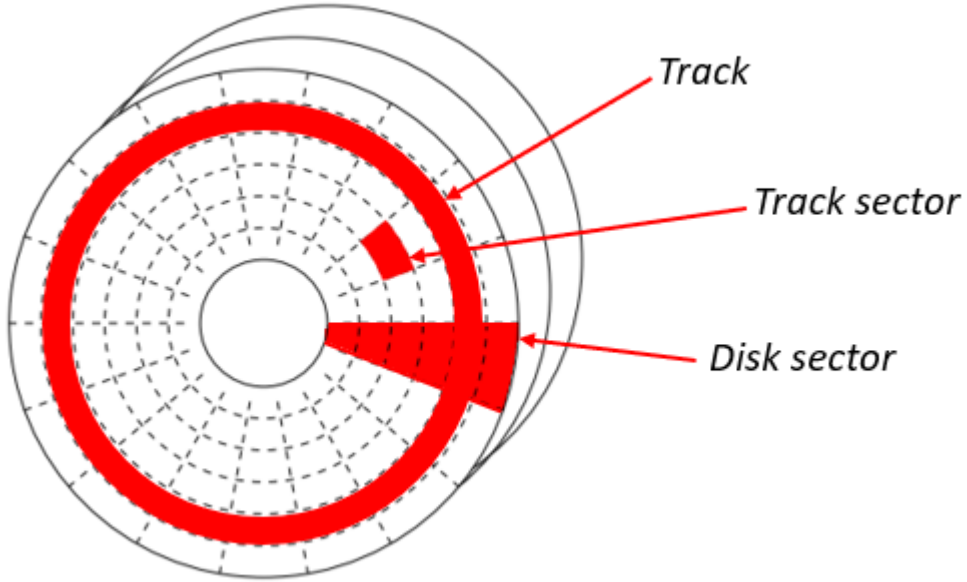
- Main form of persistent storage (at least up to now where for example SSD is taking over)
- Consist of a large number of sectors (512-byte blocks) that can be read or written
- The disk can therefore be viewed as an array of sectors
- A single 512-bit write is atomic
- Accessing blocks close to each other in the “sector array” is faster than accessing blocks far apart
- Accessing blocks in bigger chunks are faster than accessing them in a random pattern

Storage – HDD Anatomy

- Platter – Circular hard surface on which data is stored
- Spindle – holds hard drive platters in place. Is connected to a motor that spins the platter around.
- Track – a concentric circle on the surface of the platter on which data is encoded
- Disk head – reads from or writes to the disk. Attached to a disk arm that moves to the position to read from or write to.



From: https://www.computerhope.com/jargon/p/platter.htm



From: lecture "storage"

Storage – Seek Time and Rotational Delay

When reading from or writing to an HDD:

- We need to move the disk head to the right sector. This means:
  1. Moving the disk arm to the right track and
  2. Waiting for the spindle to spin the platter to the right track sector

- This takes a long time compared to the time it takes to actually read what is written on that sector.

Storage – Time Assessment

- Rotational delay – the time it takes for the platter to rotate (or spin) to the right sector
  - If the time it takes to rotate the platter one turn (the full rotational delay) is R, the average rotational delay is R/2
- Seek time – the time it takes to move the arm to the right track
  - Three faces:
    1. Acceleration as the arm gets going
    2. Coasting as the arm is moving at full speed
    3. Deceleration as the arm slows down
    4. Settling as the head is carefully positioned at the right track
- Transfer time – the time it takes to transfer data to or from the sector. This time is often insignificant compared to the rotational delay and seek time.

Storage – Exam Question

If a hard disk drive has a average seek time of 10 ms, a rotation speed of 7200 rpm (rounds per minute) ~~and a read performance of 200 MiB/s~~. Then what is the average time to read a random sector ~~on 4KiB~~

Answer:

The time it takes to read the sector is so small compared to the seek time and rotational delay so we don't have to care about it. So we just calculate the time it takes to place the disk head. This time is rotational delay + seek time.

The full rotational delay is 60000/7200 ≈ 8ms (7200/60000 = 0,12 laps per ms => 1/0,12 ms for one lap ≈8 ms), which means the average rotational delay is 8/2 = 4 ms.

The seek time is 10ms => answer is 10+4=14 ms

File Systems

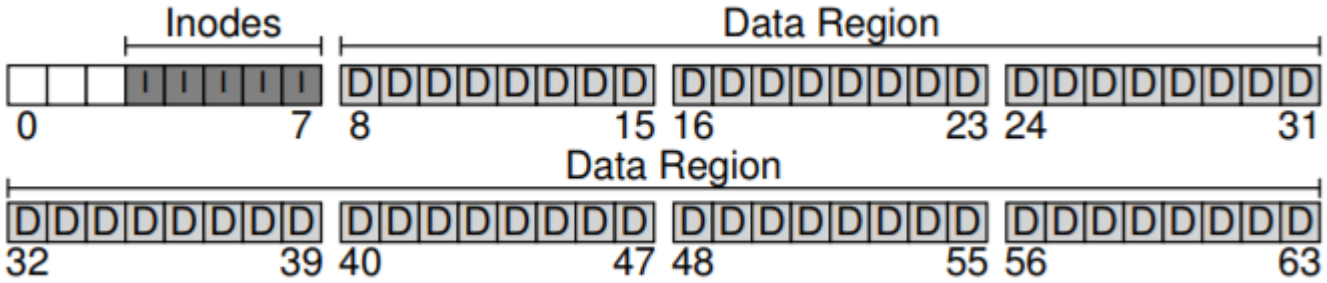
- A file system is the user space implementation of persistent storage
- A file is simply an array of bytes that can be read from or written to
- A file has a low level name called an inode number.

File Systems – Hard and Soft Link

- Hard links:
  - Created with command ln
  - Creates a file with a link to the same inode as an existing file
  - Therefore a hard link points to an inode
  - A way to represent a single file with more than one path
  - When you delete a file, it removes that file's link to the inode. We only remove one of the links. This does not affect the hard link created to that inode. Therefore, deleting the original file does not affect the file created by a hard link. Neither does renaming or moving.
- Soft (symbolic) links:
  - Created with command ln -s
  - Points to the name of a file rather than the inode
  - When deleting a target file, the link to that file becomes unusable
  - Functions kind of like a shortcut: if you move or delete the original file that the shortcut points to, the shortcut does not function properly

File Systems - Organization

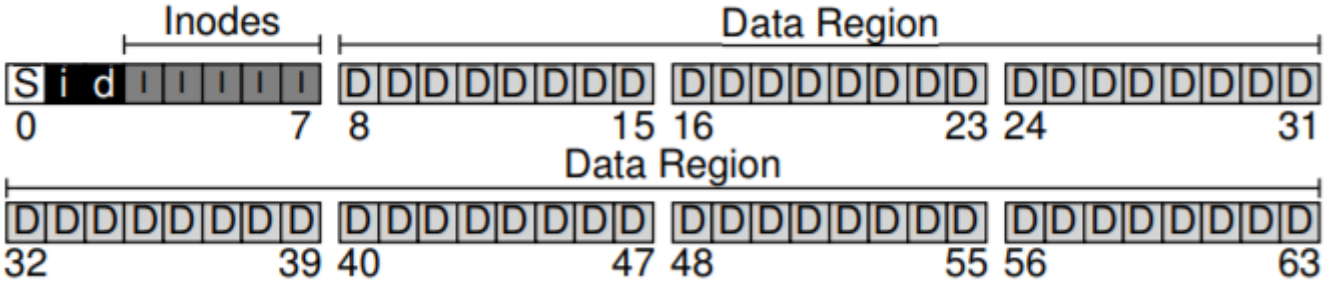
- Data blocks
  - The file system us divided into blocks containing for example user data
- Data region
  - The region of the disk where we store user data
- Metadata
  - Information about each file, for example which data blocks comprise a file, the size of a file etc
- Inode table
  - A reserved part of the disk which holds an array of inodes



From: Operating Systems: Three Easy Pieces by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau

File Systems – Organization cont.

- Allocation structure
  - Tracks whether a inodes or data blocks are free or allocated
- Bitmap
  - Allocation tracking method
  - Each bit in the structure shows whether the corresponding object/block is free (0) or allocated (1)
  - One for the inodes (inode bitmap (i)) and one for the data region (data bitmap(d))
- Superblock (S)
  - Contains information about the file system, for example how many data blocks and inodes are in the file system



From: Operating Systems: Three Easy Pieces by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau

File Systems – Inode: A Closer Look

- Each inode has a number used to refer to that inode: the inumber
- The inode contains information about a file, for example its type (e.g. regular file or directory), size, the number of blocks allocated to it, protection information, where its data blocks reside on the disk etc.
  - In other words, the inode contains the file's metadata.
- From the inumber we can get the corresponding inode

File Systems – Multi-Level Index

How should the inode refer to where data blocks are?

- A simple approach is to have direct pointers inside the inode, were each pointer refers to one disk block that refers to a file
- A problem with this is if files are very large
- To solve this, we use multi-level indices
  - Uses indirect pointers
  - Instead of pointing to a block, it points to a block that contains more pointers that in turn point to user data
  - For even bigger files, use *double* indirect pointers, and so on.

File Systems – Directories

- Directories are basically lists of {entry name, inode number} pairs
- Each directory has at least two entries: “.” that is the current directory, and “..” that is the parent directory
- Directories are treated as a special kind of file by the file system. Thus, a directory has an inode.

File Systems – Reading a File From Disk

Assume that we want to read a file /foo/bar. Assume that the file is 4KB in size. The following steps have to be taken:

1. The file system first needs to find the inode for bar from the path name by traversing the path starting from the root
2. The root has a know inode number (usually 2 in Unix systems), so the file system reads the block that contains inumber 2.
3. From the inode the file system can get pointers to the data blocks that contain the contents of the root directory. The file system can thus read through the directory looking for “foo”
4. When “foo” is found, the file system has “foo”’s inumber.
5. The file systems reads the data block containing the inode for “foo”

File Systems – Reading a File From Disk cont.

6. From the inode for “foo”, the file system reads “foo”’s directory data to find the inode number of “bar”, in the same way it found the “foo”-directory from the root.
7. The final step is to read the inode into memory. After this the file can be read.

Writing to a file is a similar process; the difference is that data blocks might have to be allocated, which makes the write operation more complex and costly compared to the read operation.

File Systems – Caching and Buffering

- Files that are frequently accessed can be cached using system memory (DRAM)
- This is very effective for read operations. For write operations however this is not as profitable as writes still have to be written to disk, which is costly
- The solution to this is write buffering
  - Instead of writing to disk every single time a write operation is issued, the writes are buffered so that they can be written to disk at the same time
  - This means fewer I/O operations to disk, and if a file is changed frequently some changes does not have to be written to disk at all (for example if a file is changed an then deleted)
  - Furthermore, the I/O operations can be scheduled to a time when it is more suitable to perform them
  - DRAWBACK: if the system crashes before a batch of writes are performed, more updates are lost. Because of this, writes to memory can be forced if one does not want to take this risk

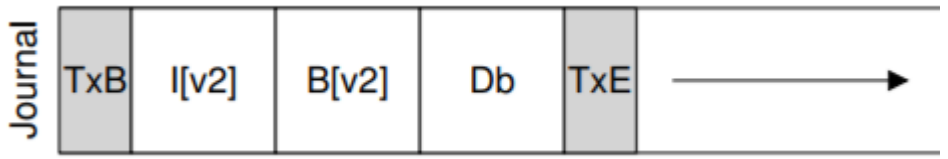
File Systems – Journaling: Basic Idea

One way to solve the problem if lost data in the case of a crash. Also called “write-ahead logging”

- Idea: when writing to the disk, before performing the update, write a note on the disk describing what is about to be done. In this case, if the system crashes during the write, the system can look at the note an try again.
- By this the system can know what to fix and how
- Makes updating a structure on the disk take longer but reduces the time it would take to recover the data.

File Systems – Journaling: How it's Done

- The log consists of five blocks:
  - TxB: Transaction begins block. Contains some information about the update, like the final destination address of the content blocks and a transaction identifier (TID)
  - Content blocks: what should actually be written to the disk. These are three blocks: data block (the user data), the inode and the bit maps
  - TxE: Transaction ends block. Informs the system that the transaction ends here and that there is no more data to write to the disk. Also contains the TID.
- Journaling has two steps to write the data to disk:
  1. Journal write: write all five blocks to the log and wait for the write to complete
  2. Checkpoint: Write the three content blocks to their final destinations on the disk



From: Operating Systems: Three Easy Pieces by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau

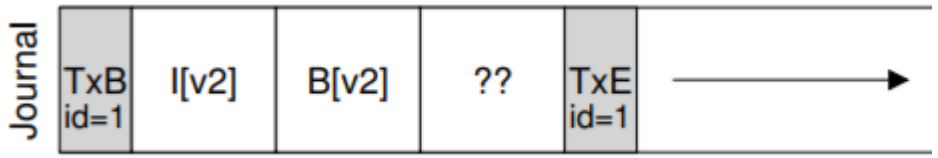
File Systems – Journaling: The Elephant in the Room

But what happens if the system crashes while writing to the log??

- Well, if the journal is incomplete we haven't made any real writes. The operations is as if it was never performed.
- This means we will not have any inconsistent states where only some writes have taken place. I.e. we have atomicity: either all writes will be performed (if the journal is complete so that we know what work to perform) or no update is made (if the journaling fails)

Great! But we might still have a problem:

- If we perform all writes to the journal at once, there is no guarantee that they will be in order. So we might actually write the TxE block before all content blocks are written to the journal. So the system will think that we have a complete journal when in fact we do not.



From: Operating Systems: Three Easy Pieces by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau

File Systems – Journaling: Final Solution

- Super easy solution: just write one block at a time: write TxB, wait for it to complete, then write the first content block, wait for it to complete etc.
- Yes, this solves the problem, but it is VERY slow!
- Better solution! Write the first four blocks, wait for them to complete, then write the TxE block. This way we guarantee that all content blocks will be in the log before we write the TxE block. And if the TxE block is not completely written, we know that something went wrong.

File Systems – Exam Question

Assume that we have a simple file system without journaling where we write directly to inodes, bit maps and data blocks. Also assuem that we have a crash and the we when creating a file only are able to do two of the three required updates. In each of the cases below, explain the problem we will have.

- inode and bitmaps
- bitmaps and data blocks
- inode and data blocks

File Systems – Exam Question Answer

Answer: When updating a file we need to update the inode, bit maps and data blocks. The following scenarios describe what happens if only two of these three updates are made

1. Inode and bitmaps

   The system metadata is consistent, but the data block (that contains the data) is not up-to-date, which means that when we try to access the block we will access the wrong data.

2. Bitmaps and data blocks

   Even though the bitmap indicates that the block is used and the data is actually in the data block, we cannot know what file to access as we do not have any inode that points to the data block. This way the data is lost as we cannot access it.

3. Inode and data blocks

   The file exists, but if the bit maps do not indicate that the inode and data block are used it might allocate other things in this space, which would mean the data is lost.

Virtualization

- Ability to run several operating systems on top of each other
- Can done by using:
  - Virtual machine monitors a.k.a Hypervisors
  - Containers
  - Emulators

Virtualization - Hypervisor

- Used when we want to run different operating systems
- Both operating systems think they are in control of the hardware but the hardware is actually controlled by the hypervisor and the resources are shared among the operating systems
- There are two types of hypervisors
  - One that runs directly on the hardware
  - One that runs on top of the operating system, ex: VirtualBox

Virtualization – Container & Emulator

- Container
  - An environment that is delimited from the rest of the systems
  - Runs on the same computer with its own memory. Filesystem, network etc
  - They are limited to use the same operating systems
  - Ex: Linux containers
- Emulator
  - Emulates different hardware than the host machine
  - Ex: QEMU

Virtualization – Exam Question

Describe one important reason why you would like to run several virtual operating systems instead of one single.

Answer: For example if you need to be able to use functions that are available on different versions of an operating system or completely different operating systems. using for example hypervisors or emulators we can actually run several operating systems on one machine instead of having different machines with different operating systems