1. a) Trap instruction is run in user mode, and privileged operation is run in kernel mode

Notes

- **Previliged Instructions**
  - Is the instruction that can run only in **kernel mode**
  - Attempt at execution in **user mode** → treated as an illegal operation & will not run.
- **Trap**
  - Is a special hardware instruction
  - Is a software generated interrupt [4]
  - Is a type of synchronous interrupt [1]
  - Is caused by an exceptional condition [1]
    1. Division by zero [1]
    2. Invalid memory access (segmentation fault) [1]
    3. Previleged instruction by **user mode** code [2]
  - Usually results in a switch to **kernel mode** → Operating system performs action → Returns control to oroginal process
- **Trap Instruction**
  - Is executed when a user wants to invoke a service from the operating system (i.e. reading hard drive) in **user mode**
  - Raise (the processor) privilege level to kernel mode
- **User Mode**
  - Is restricted
  - Executing code has no ability to *directly* access hardware or reference memory [3]
  - Crashes are always recoverable [3]
  - Is where most of the code on our computer / applications are executed [3]
- **Kernel Mode**
  - Is previleged (non-restricted)
  - Executing code has complete and unrestricted access to the underlying hardware [3]
  - Is generally reserved for the lowest-level, most trusted functions of the operating system [3]
  - Is fatal to crash; it will halt the entire PC (i.e the blue screen of death) [3]

References

1) Wikipedia, Trap (computing), link
2) University of Utah, CS5460: Operating Systems Lecture 3 - OS Organization, link
3) Coding Horror, Understanding User and Kernel Mode, link

4) ETH Zurich, Programming in Systems, link

b) No. Lock uses a variable with binary states 0 (acquired) and 1 (available), where as semaphore uses counter variable that can have value greater than 1 to keep track of the amount of resource remaining.

Notes

- **Locks**
  - Is a variable with two boolean states
    - 1 - (available/unlock/free)
    - 0 - (acquired/locked/held)
  - Has two operations
    1. `acquire()`

```
boolean test_and_set(boolean *lock)
{
      boolean old = *lock;
      *lock = True;
      return old;
}
boolean lock;

void acquire(boolean *lock) {
       while(test_and_set(lock));
}
```

    2. `release()`

```
void release(boolean *lock) {
       *lock = false;
}
```

  - Is put around critical section to ensure critical section executes as if it's a single atomic instruction

```
1  lock_t mutex; // some globally-allocated lock 'mutex'
2  ...
3  lock(&mutex);
4  balance = balance + 1;
5  unlock(&mutex);
```

  - Can only be released by the thread that acquired it
  - Is used to protect shared resource (e.g. from race condition in files and data structure) [2]
- **Semaphore**
  - Is an abstract data types suitable for synchronization problems [2]
  - Has variable `count` that allows arbitrary resource count [1]
  - Has two atomic operations
    1. (`wait/P/decrement`) - block until `count > 0` then decrement variable

```
wait(semaphore *s) {
        while (s->count == 0) ;
        s->count -= 1;
}
```

    2. (`signal/V/increment`) - increment `count`, unblock a waiting thread

```
signal(semaphore *s) {
        s->count += 1;
        ......                  //unblock one waiter
}
```

  - Can be signaled by any thread [2]

References

1) Wikipedia, Semaphore (programming), link
2) Stack Overflow, Difference between binary semaphore and mutex, link

c) If both access are read, then concurrency error will not occur.

Notes

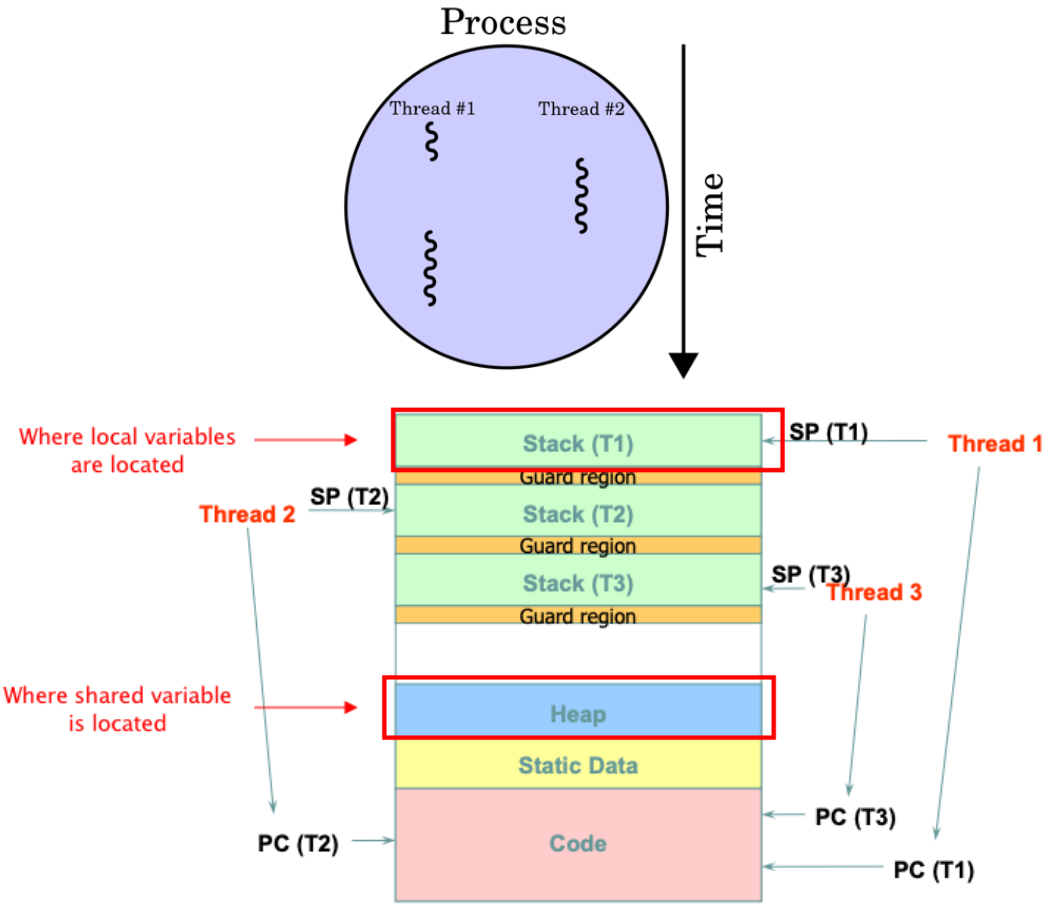
- What is concurrency error? Where and when does it occur?
- **Concurrency**
  - Is the ability of different parts or units of a program, algorithm, or problem to be executed out of order, without affecting the final outcome. [1]
- **Concurrency Error**
  - Two types of concurrency errors [3]
    1. **Deadlock:** A situation wherein two or more processes are never able to proceed because each is waiting for the others to do something

       Key: Circular wait

    2. **Race Condition:** a timing dependent error involving shared state

  - **Data Race:** Concurrent accesses to a shared variable and at least one access is a write
  - **Atomicity Bugs:** Code does not enforce the atomicity programmers intended for a group of memory access
  - **Order Bugs:** Code does not enforce the order programmers intended for a group of memory access

- **Thread**
  - Is the smallest sequence of programmed instructions that can be managed independently by a schdeduler [2]



  - A thread is bound to a single process
  - A process can have multiple threads

References

1) Wikipedia, Concurrency (computer science), link
2) Wikipedia, Thread, link
3) Columbia University, Concurrency Errors, link

d) No, limited execution limits what a process can do without OS assistance by restricting access to hardwares, setting up interrupt timer, and trap handlers.

Notes

- **Virtualization of CPU**
  –
- **Limited Direct Execution**
  – Idea: Just run the program you want to run on the CPU, but first make sure to set up the hardware so as to limit what process can do without OS assistance
  – baby proofs the CPU by

    1. Setting up trap handlers
    2. Starts an interrupt timer
    3. Run processes in a restricted mode

    Example

    Baby proofing a room:

    * Locking cabinets containing dangerous stuff and covering electrical sockets.
    * When room is readied, let your baby roam free in knowledge that all the dangerous aspect of the room is restricted
- **Trap Handlers**
  – Is instruction that tells the hardware what to run when certain exceptions occur

    Example

    What code to run when

    1. Hard disk interrupt occurs
    2. Keyboard interrupt occrs
    3. Program makes a system call?
- **Timer Interrupt**
  – Is a hardware mechanism that ensures the user program does not run forever
  – Is emitted at regular intervals by a timer chip [1]

References

1) Wikibooks, Operating System Design/Processes/Interrupt, link

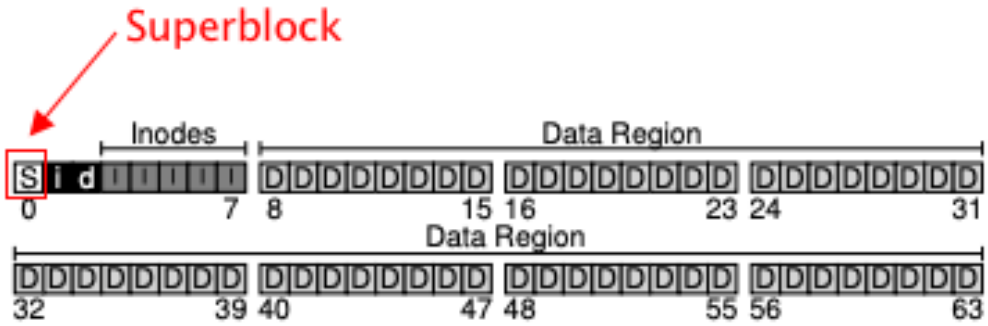
e) No, alhtough data blocks are fragmented, it doesn't suffer from external fragmentation because indexed file system has pointers in inodes that points to each data block.

Notes

- **Index-based File System**

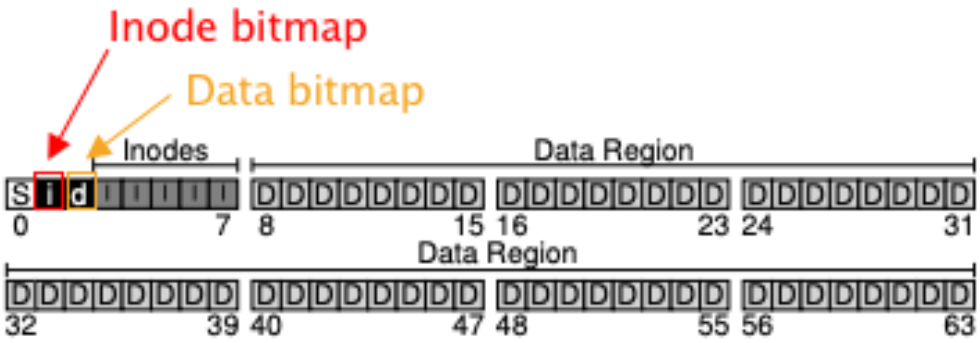
  - Has following parts
    - Superblock
    - Inode Bitmap
    - Data Bitmap
    - Inodes
    - Data Region
  - Each block in file system is 4KB
  - Uses a large amount of metadata per file (especially for large files)

- **Superblock**

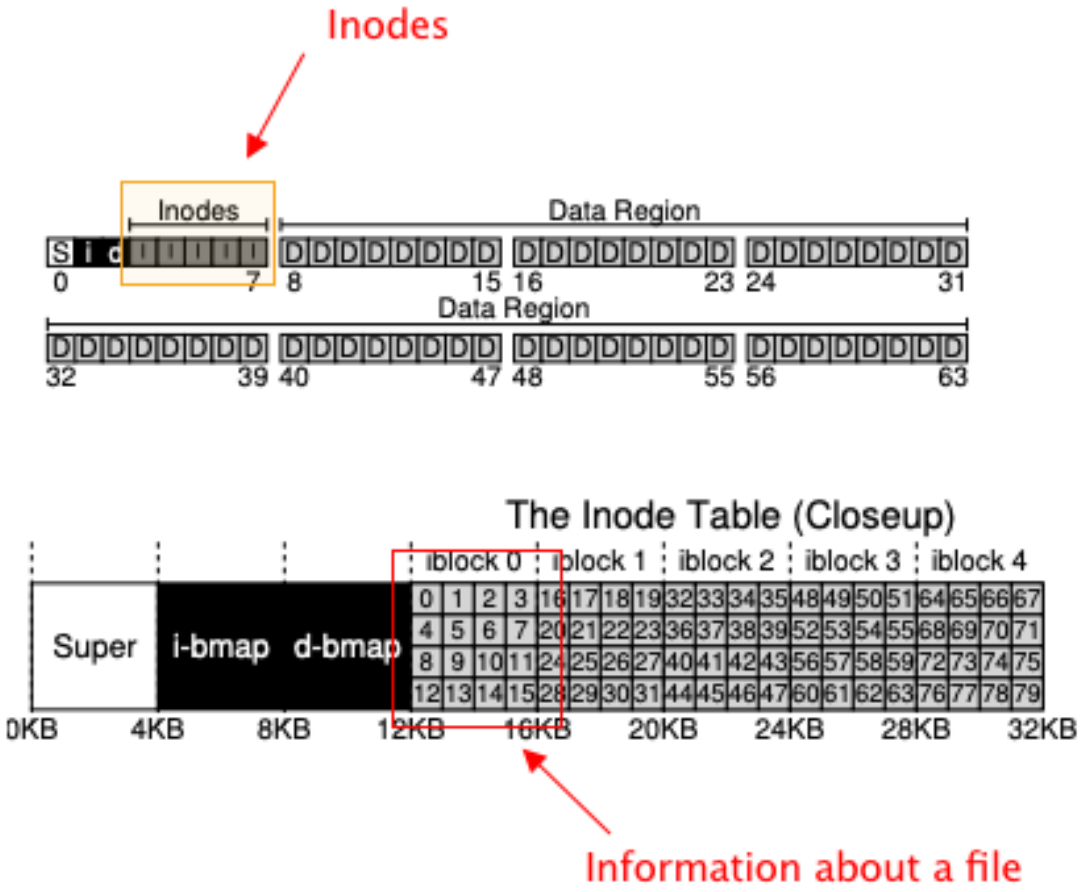


  - contains information about the file system, including
    1. the number of inodes and data blocks in a particular file system
    2. the magic number of some kind to identify the file system type (e.g NFS, FFS, VSFS)
  - The OS reads superblock first to initialize various parameters, and then attach volume to the file-system tree

- **Bitmap**



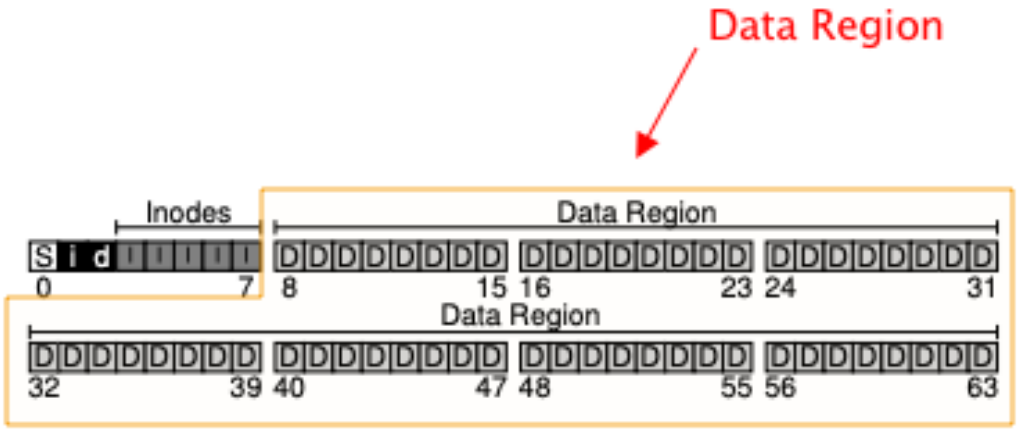
  - Tracks whether inode or data blocks are free or allocated
  - Is a simple and popuar structure
  - Uses each bit
    * `0` means free
    * `1` means in use
  - **Data Bitmap** is bitmap for data region
  - **Inode Bitmap** is bitmap for inode region
- **Inode**



  - Is a short form for **index node**
  - Contains disk block location of the object's data [1]
  - Contains all the information you need about a file (i.e. metadata)
    * File Type
      · e.g. regular file, directory, etc
    * Size
    * Number of blocks allocated to it
    * Protection information
      · such as who owns the file, as well as who can access it
    * Time information

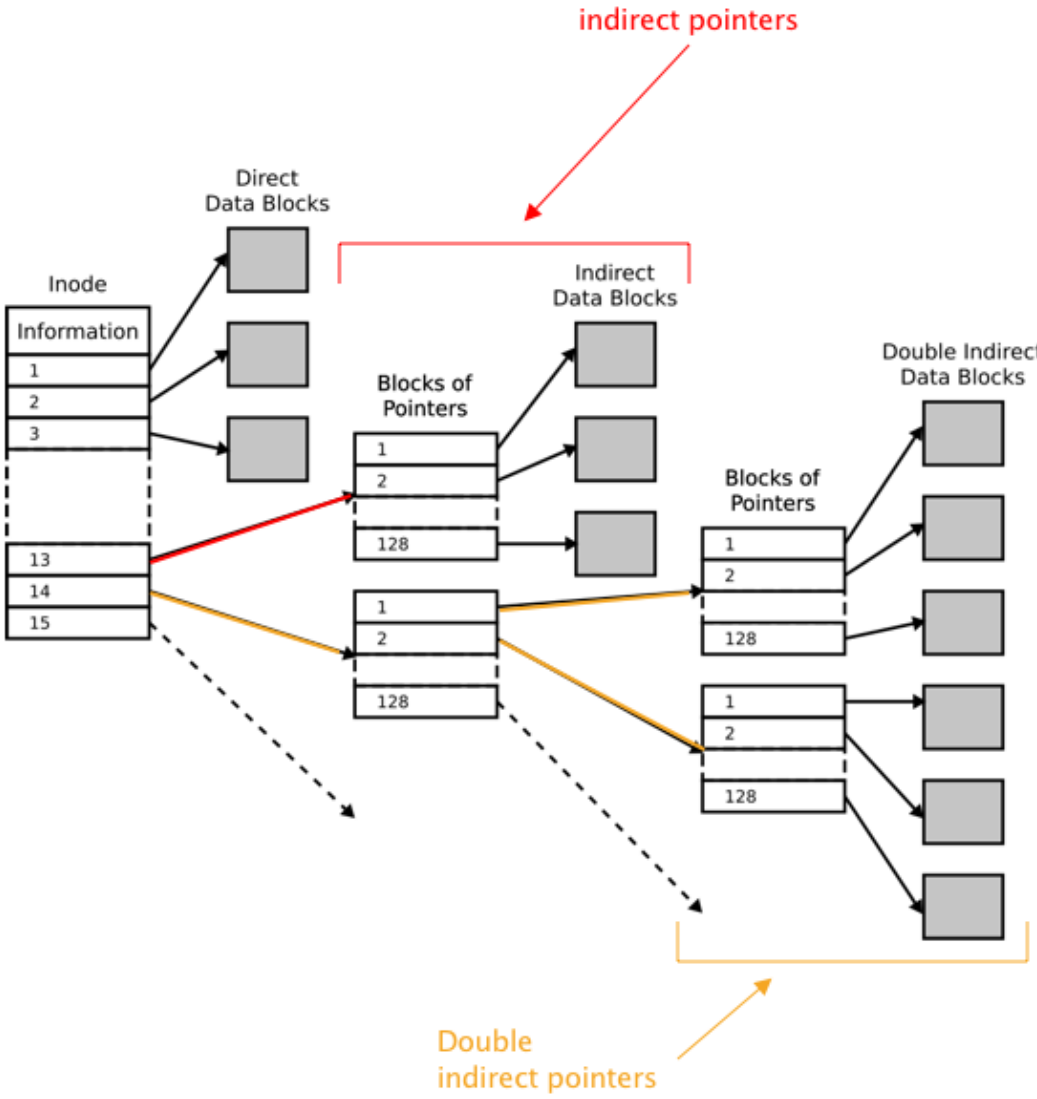
        - e.g. When file was created, modified, or last accessed
    - Location of data blocks reside on disk

- **Data Region**



  - Is the region of disk we use for user data

- **Multi-Level Index**



  - Is for supporting bigger files
  - Uses **indirect pointer** in inode that points to more pointers
  - Uses **double indirect pointer** for even larger files
    - is a pointer in inode that points to a block that contains pointers to indirect blocks

- **External Fragmentation**
  - Is various free holes that are generated in either your memory or disk space. [2]
  - Are available for allocation, but may be too small to be of any use [2]
- **Internal Fragmentation**
  - Is wasted space within each allocated block [2]
  - Occurs when more computer memory is allocated than is needed

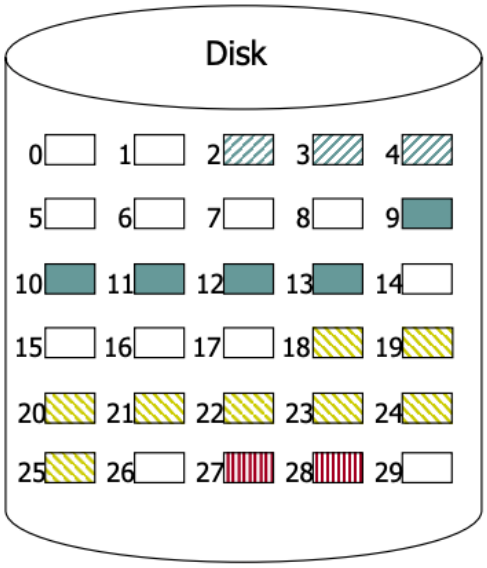
References

1) Wikipedia: inode, link
2) Washington University, Explain the difference between external fragmentation and internal fragmentation, link
3) Wikipedia, Inode pointer structure, link

f) No, extent-based file system requires less disk block access, because unlike index-based file system (which consumes disk space for many indirect pointers in addition to data blocks for large files), extent-based file system only requires one pointer plus contiguous data blocks.

Notes

- **Extent Based File System**



directory

File Name	Start Blk	Length
File A	2	3
File B	9	5
File C	18	8
File D	27	2

  - Is simply a disk pointer plus a length (in blocks)
    - Together, is called **extent**
  - Often allows more than one extent
    - resolve problem of finding continuous free blocks
  - Is less flexible but more compact
  - Works well when there is enough free space on the disk and files can be laid out contiguously

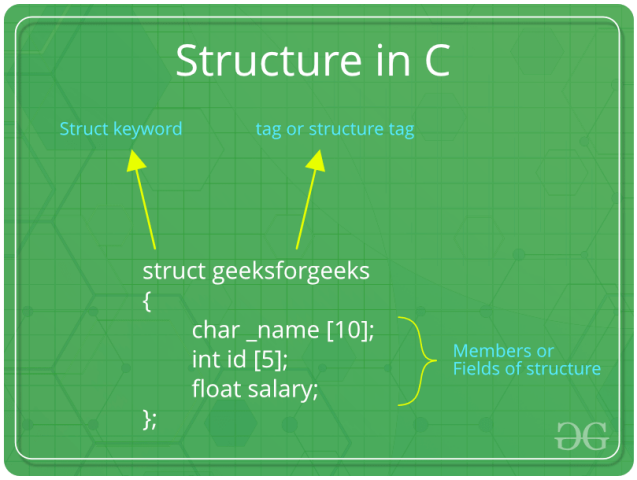
Example

Linux's ext4 file system

2. a) 1) Process State
   2) Process Number
   3) CPU scheduling information / Process Priority

   Notes

   - **Fields**
     - Is the members in a structure



   - **Process List**
     - Is a data structure in kernel or OS
     - Contains information about all the processes running in the system
   - **Process Control Block**
     - Is a data structure in kernel or OS
     - Contains all information about a process
     - Is where the OS keeps all of a process' hardware execution state
     - Generally includes
       1. Process state (ready, running, blocked)
       2. Process number
       3. Program counter: address of the next instruction
       4. CPU Registers: is saved at an interrupt
       5. CPU scheduling information: process priority
       6. Memory management info: page tables
       7. I/O status information: list of open files

   b) Context switch switches from the current process to a different one. To achieve this, all resource information about process needs to be saved, including:

   - Process Number
   - Process State
   - Process Counter,

- CPU Registers
- Memory management info
- I/O status Information

Notes

- I am reading that process needs to save resource information before context switch. I need to verify with professor regarding this information.
- **Context Switch**
  - is the process of storing the state of a process or thread, so it can be restored and resume execution at a later point [1]
  - happens during a timer interrupt or system call
  - process context switch needs to save the following [2]
    - Process Number
    - Process State
    - Process Counter,
    - CPU Registers
    - Memory management info
    - I/O status Information
  - thread context switch needs to save the following
    - Process Counter,
    - CPU Registers
  - May hinder performance

References

1) Wikipedia: Context switch, link
2) University of Washington, CSE 451, Spring 2000 Solutions to Homework 2, link

c) One concrete example is CPU virtualization.

In this case, CPU is virtualized, and OS achieves CPU virtualization by

1) Limiting the process can do without OS assistance by setting up trap handler and starting an interrupt timer
2) Intervening at key points in time to perform privileged operations and switch out processes/operations that monopolized the CPU too long

Notes

- **Virtualization**
  - is an act of creating an illusion that there are as many hardware resource as each program needs.

- **Virtualizing CPU**
  - Turns a single CPU into a seemingly infinite number of CPUS, and allows many programs to seemingly run at once
  - To implement CPU virtualization, the OS needs low-level machinery called **mechanism** and high level intelligence called **policies**
  - Steps
    1. Involve OS to setup hardware hardwre to limit what the process can do without OS assistance (**Limited Direct Execution**)

       This is done so by

       1. Setting up trap handler
       2. Starting an interrupt timer (so process won't last forever)
    2. Involve OS to intervene at key points to perform previleged operations or switch out operations when they have monopolized the CPU too long

- **Limited direct execution**
  - Is a mechanism of CPU virtualization
  - Baby proofs the CPU by setting up hardwre to limit what the process can do without OS assistance
  - Runs program in CPU once baby proofed
  - Does so by
    * Setting up trap handler
    * Starting an interrupt timer (so process won't last forever)
- **Scheduling policies**
  - Are series of polices of CPU virtualization
    * **First In First Out**
    * **Shortest Job First**
    * **Shortest Time-to-completion First**
    * **Round Robin**
    * **Multi-level Feedback Queue**
- **Virtualizing Memory**
  - Basic Idea: For the most part, let the program run directly on the hardware; however, at certain key points in time (e.g. system call, timer interrupt), arrange so that the OS gets involved and make sure the 'Right' thing happens.
  - Like CPU, many programs are sharing the memory at the same time
  - Like CPU, the goal is to create an illusion that it has its own code and data
  - Like CPU, the memory also needs low-level machinery called **mechanism**, and high level intelligence called **policies**
  - Steps

1. Use **address translation** to transform each memory access, changing **virtual address** provided by instruction to **physical address**
   * Memory access includes instruction fetch, load, or store
   * Is done using hardware
2. Invove OS at key points to **manage memory**

   Memory management includes

   1. Setting up hardware so correct translations take place
   2. Keeping track of which locations are free and which are in use
   3. Judiciously intervening to maintain control over how memory is used

- **Address Translation**
  - Is also called **hardware-based address translation**
  - Is a mechanism of memory virtualization
  - Is the technique of transforming virtual address to physical address

3. `a)`, `b)` and `c)` are true

   Notes

   - I need clarification from professor on the meaning behind 'Mutex is held?'

   Correct Solution

   `c)` and `d)` are true

4. **Notes**

   - **Block**
     - Size of each block is 4KB
   - **lseek** s
     - **Syntax:** `off_t lseek(int fildes, off_t offset, int whence)`
       * `fildes` - file descriptor
       * `offset` - file offset to a particular position in file
   - **Kilobyte**
     - 1 kilobyte is 1024 bytes
   - **file**

- is an array of bytes which can be created, read, written and deleted
- low-level name is called **inode number** or **i-number**

- **Reading a File From Disk**

  Example

  When

  ```
  open("/foo/bar", O_READONLY)
  ```

  is called

  - the goal is to find the inode of the file `bar` to read its basic information (i.e. includes permission, information, file size etc)
  - done by traversing the pathname and locate the desired inode
  - Steps
    1. Begin traversal at the root of the file system, in the **root directory**
    2. Find **inode** of the root directory by looking for `i-number`
       * **i-number** is found in it's parent directoy
       * for root directory, there is no parent directory
       * it's inode number is 2 (for UNIX file systems)
    3. Read the **inode** of root directory
    4. Once its **inode** is read, look inside to find pointers to data blocks
    5. Recursively traverse the pathname until the desired inode is found (e.g `foo` $\rightarrow$ `bar`)
    6. Issue a `read()` system call to read from file
       * `fd` with offset `0` reads the first file block (e.g. `bar data[0]`)
       * `lseek(..., offset_amt * size_of_file_block)` is used to offset/move to desired block in `bar`
    7. Trasnfer data to `buf` data block
    8. Close `fd`. No I/O is read.

- **The Multi-Level Index**
  -