HOMEWORK QUESTIONS

1. Explain how paging works. In this explain the role of PTBR. Which parameters determine the size of page table in bytes, and number of entries in it? How is a 32-bit address split into parts, and how is each part used?

2. Explain the code of setupkvm(). Which logical addresses are mapped by setupkvm() and to which physical addresses?

-> 1. Kalloc //gets 4b mem and allocates
   2. Memset //make full pg 0 so prev data is erased
   3. Kvmalloc //Iterate over kmap array. iterate 4 times. Creates pgtable for each separately
      Since each has diff permissions.
   4. Kmap
   5. map pages // creates pages
   6. Readi //reads and puts in mem
   7. Error checks
   8. Allocuvm //creates user pddir, table
   9. Loaduvm //loads data in pages

3. How does the xv6 kernel obtain the parameters passed by user-application to the system call? E.g. how does kernel obtain the paramters to read(fd, &ch, 1); ?

Ans: Accesses them off of **user** stack (lol).
Detailed Ans: User programs push the args on the process' user stack. Execution of system call is done in the kernel mode hence the stack is changed to kernel stack. In order to access the arguments the user stack's esp is accessed through the trapframe. Rest you can read in the code.

4. Explain the meaning of each field in struct proc. Draw a nice diagram of struct proc with as many details as possible. What are the contents of the trap frame - draw a neat detailed diagram. What is the 'chan' field?

-> 1. size
   2. Pgdir //points to page dir of process. Page comes from kalloc. Gives pointers to both user
      and kernel stack
   3. Kstack //points to page obtained from kalloc. This is sthe kernel stack. Trap frame hosted
      on kernel stack
   4. Proc * parent
   5. Tf //trapframe pointer. Points to specific location on kernel stack
   6. Context //pointer stores context [eip=forkret(),ebp,ebx,esi,edi]
   7. Chan // used when process is in wait queue for some i/o operation. Basically an array
which stores address of variables indicating what the proc is waiting for.
   8. Int killed //=1 if process is killed
   9. File * ofile //points to array of open file descriptors outside struct proc
   10. Inode * cwd //pointer to inode of cwd of processes. Getcwd setcwd Manipulate inode
   11. Name //array with name of processes

5. Explain the mappages and walkpgdir functions.
->
   1. Map pages calls walkpgdir
   2. Walkpgdir creates page table mapping (first address is kernbase i.e. 2GB)
   3. After pg table allocated, exec pushes args on stack
   4. Sets up args to main (argc, argv)
   5. Saves context of prev process //refer line 97 of exec code in xv6
   6. Calls switchuvm() //switches cr3 here

6. What are the different usages of a semaphore? Give one example each.

Semaphores are a fundamental synchronization mechanism used in operating systems for coordinating access to shared resources among multiple processes or threads. They help prevent race conditions and ensure that critical sections of code are executed atomically

1. Binary Semaphore: A binary semaphore is a semaphore with only two possible states, often used for mutual exclusion or as a signaling mechanism.
Example: In a producer-consumer problem, where multiple producers produce items and multiple consumers consume those items, a binary semaphore can be used to synchronize access to the shared buffer. Producers wait on the semaphore when the buffer is full, and consumers wait when the buffer is empty.

2. Counting Semaphore: A counting semaphore is a semaphore with an integer value that can range over an unrestricted domain.
Example: In a scenario where a limited number of resources (e.g., printers, database connections) are available, a counting semaphore can be used to control access to these resources. Each time a resource is acquired, the semaphore's count decreases, and when a resource is released, the count increases. Processes or threads can block if the count reaches zero, indicating that no resources are available.

3. Mutex Semaphore: A mutex (short for mutual exclusion) semaphore is a specialized binary semaphore used to enforce mutual exclusion on a critical section of code.
Example: In a multi-threaded environment, if multiple threads need to access a shared resource (e.g., a global variable), a mutex semaphore can be employed to ensure that only one thread can access the resource at a time. Threads attempting to acquire the mutex will block if it's already held by another thread, preventing concurrent access and potential data corruption.

4. Named Semaphore: Named semaphores are semaphores that have a unique name in the operating system's namespace, allowing processes to share synchronization primitives.
Example: In inter-process communication (IPC), named semaphores can be used to coordinate access to shared memory regions or file resources. Processes can create or open a named semaphore by its unique name and use it to control access to the shared resource, enabling synchronization across multiple processes

7. Explain the code of exec(). Which are the major functions called by exec() and what do they do? What does allocproc() do?
->

1. Exec -> setupkvm -> kvmalloc -> map pages -> walkpgdir -> readi -> allocuvm -> loaduvm.
2. If page is not present at a given location then map pages calls alloc().
3. Allocuvm only allocates pgdir, table (not data)

4. Allocproc finds the first unused process in ptable and allocs it a pid and space for context and trapframe

8. Write one program that deadlocks, and one that livelocks.

9. Which data structures on disk are modified if you delete a file, say /a/b, on an ext2 file system? Try to enlist considering the worst possibilities.

1. Inode Table: The inode corresponding to the file /a/b will be modified to mark it as free. If the inode is part of an inode table block, that block might need to be modified to reflect the change in the inode's state.

2. Directory Entry: The directory entry for file b in directory /a will be modified to remove the reference to the inode of b. If the directory block containing this entry becomes empty after the deletion, it might be marked as free, and the corresponding data block might be modified to reflect this change.

3. Bitmaps: The block bitmap and inode bitmap might be updated to mark the blocks and inodes that were previously allocated to file b as free for reuse.

4. Data Blocks: If file b has any data blocks allocated to it, these blocks will be marked as free in the block bitmap, and the corresponding data blocks might be modified to reflect this change.

10. How will you write code to read a directory from an ext2 partition ? How will the entries in the directory be read by your code so that you can list all files/folders in a directory ? What is rec_len and how it is used in creating/deleting files/folders?

11. Compare concurrency and parallelism.
Concurrent and parallel programming are related but not synonymous concepts. Concurrent programming is more about the logical structure and behavior of programs, while parallel programming is more about the physical execution and optimization of programs. Concurrent programs can run on single-core or multicore processors, while parallel programs require multicore or distributed systems. Concurrent programs can be parallel, but not all concurrent programs are parallel. For example, a web server that handles multiple requests concurrently may not run them in parallel if it has only one processor. Similarly, parallel programs can be concurrent, but not all parallel programs are concurrent. For example, a matrix multiplication that runs on multiple cores may not be concurrent if it does not have any coordination or synchronization between them.

12. Which are the different symbols in the 'kernel' ELF file, used in the xv6 code ? (e.g. 'end')
In the xv6 kernel ELF file, there are several symbols used to mark different parts of the code. Some of the common symbols found in the kernel ELF file include:

1. **_start**: The entry point for the kernel code.
2. **_etext**: Marks the end of the text (code) section.
3. **_edata**: Marks the end of the data section.
4. **_end**: Marks the end of the kernel image.
5. **_binary_obj_name_start**, **_binary_obj_name_end**: Symbols used to denote the start and end of binary objects included in the kernel image, such as initcode, bootblock, etc.
6. **_rodata_start**, **_rodata_end**: Marks the start and end of the read-only data section.

These symbols help in understanding the layout and organization of the kernel image in memory. They are often used in linker scripts and other parts of the build process to properly place code and data sections in memory.

13. Which memory violations are detected in xv6 ? Which violations are not detected?
In xv6, a simple Unix-like operating system developed for educational purposes, memory violations are not extensively detected by default. However, some common memory violations that might be detected include:

1. **Null Pointer Dereference**: Accessing or dereferencing a null pointer. This can sometimes lead to a kernel panic, but xv6 does not always detect this explicitly.

2. **Out-of-Bounds Memory Access**: Attempting to access memory outside the bounds of an allocated region, such as accessing an array element beyond its size.

3. **Invalid Memory Access**: Trying to access memory that is not mapped or accessible.

4. **Double Free**: Freeing memory that has already been freed, leading to potential memory corruption.

However, xv6 does not provide comprehensive protection against all types of memory violations. For example, it may not detect:

1. **Use-after-Free**: Accessing memory after it has been freed. This can lead to undefined behavior but may not always be explicitly detected by the system.

2. **Buffer Overflows**: Writing more data into a buffer than its allocated size. xv6 does not typically have built-in protections against buffer overflows.

3. **Memory Leaks**: Failing to deallocate memory after it's no longer needed. xv6 does not have built-in mechanisms to detect or handle memory leaks.

Overall, while xv6 may catch some basic memory violations, it is not designed to be a robust environment for detecting and handling all possible memory-related issues. It's primarily intended for educational purposes and to provide a basic understanding of operating system principles.

14. explain in 1 line these concepts: mutex, spinlock, peterson's solution, sleeplock, race, critical section, entry-section, exit-section.

15. How do sched() and scheduler() functions work? Trace the sequence of lines of code which run one after another, when process P2 gets scheduled after process P1 enconters timers interrupt.

16. Explain how the spinlock code in xv6 works.

17. Explain the meaning of as many possible fields as possible in superblock, group descriptor and inode of ext2.
-> In the ext2 filesystem, the superblock, group descriptor, and inode contain crucial information about the file system structure and organization:

1. **Superblock**: The superblock is a data structure at the beginning of the filesystem that contains essential information about the filesystem, including:
   - Filesystem type (ext2)
   - Total number of inodes and blocks
   - Block size and fragment size
   - Blocks per group and inodes per group
   - First non-reserved inode and inode size
   - Mount time and last write time
   - Filesystem state (clean or dirty)
   - Error handling information

2. **Group Descriptor**: The group descriptor is a data structure that exists for each block group in the filesystem. It contains information specific to each group, such as:
   - Block bitmap location and size
   - Inode bitmap location and size
   - Inode table location and size
   - Free block and inode counts
   - Directory count
   - Last mounted time
   - Filesystem state
   - Group checksum (in newer versions like ext3 and ext4)

3. **Inode**: An inode is a data structure that represents a file or directory in the filesystem. Each file or directory has a unique inode, which contains metadata about the file, such as:
   - File type and access permissions
   - Owner and group
   - File size and timestamps (creation, modification, and access)

  - Pointers to data blocks (direct, indirect, and doubly indirect)
  - Number of hard links
  - File flags and attributes
  - File system-specific attributes (e.g., extended attributes)
  - Access control lists (ACLs) and capabilities

These structures are crucial for the functioning and organization of the ext2 filesystem, providing information about the layout of data on disk, allocation of blocks and inodes, and metadata associated with files and directories.

18. How can deadlocks be prevented?
-> Write code in such a way that it will invalidate one of the 4 conditions necessary for deadlock i.e. mutual exclusion, hold & wait, no preemption, cyclic wait (most used).
1. Locking hierarchy (used in kernels): keep an eye on all locks code is holding at any given point in time.
2. Lock ordering (used in xv6): priority given in ppt 16, slide 77.

19. Explain the code of swtch() by drawing diagrams.

LAB TASKS

1. What is a Zombie process? is there a difference between an orphan and a zombie process?

2. Does the Demo given by Abhijit always create a Zombie process?

3. True/False? Every dead process first become a Zombie.
-> True

4. How is a Zombie process "cleared"? (Observe first that it gets cleared)

5. Write a C code and commands to show use of dup() using /proc?

6. How is a Zombie process different from Zombies in movies ?
-> init adopts zombie processes and eventually they finish.

7. Write a small code and commands, to show use of pipe() using /proc.

8. List the names of application programs that are part of the xv6 git repo.

9. Which is the complete "qemu" command used to run xv6, by the Makefile?

10. Suppose you want to run xv6 by specifying one more extra disk. How will you do that ?

11. What happens when you run xv6 (using qemu) with 2G RAM? Anything special or just normal execution?

15. Can you list all commands used to compile the "_ls" program in xv6 project?

16. What are all those options provided to gcc, to compile an application like _ls? what is -static? what is -m32?

17. What will happen if you change the entry in "gdtdesc" to be base=0 and limit = 2 GB?
Ans: It works fine as kernel has absolute addressing from 0x7c00 onwards, hence 2Gb limit is enough

18. Is the line movw %ax, %es really necessary in bootasm.S ?
Ans: no, as es register is never used in xv6 (see the output of objdump on bootasm.S). Similarly mov ax,dx is also not needed as dx is by default 0

19. Which all parts of the code actually push something on the stack starting at 0x7c00 and what do they push ?

20. How many program headers are there in the kernel and what do they mean?
Ans: 3 program headers in the Kernel. Maybe : First one is CS, second is Data seg and last is stack seg

21. readseg((uchar*)elf, 4096, 0); will read the first 4k bytes from the xv6.img into memory. This should include the bootloader also. Isn't it? If yes, how is "elf" pointer correct?
Ans: No readseg, gives an offset to make the sector read from =1 whilst translating bytes to sectors . Refer bootmain.c → readseg function

22. Why V2P_WO is used in entry.S and not V2P ?
Ans: Dumb answer but: No typecasting is required in entry.s. Perhaps due to the fact that in assembly no explicit typecast to uint is required. In fact during pre-processor directives the ‘uint’ symbol is unrecognized by the assembler.

23. What will be the effect if we remove the 0th entry from entrypgdir Array ?

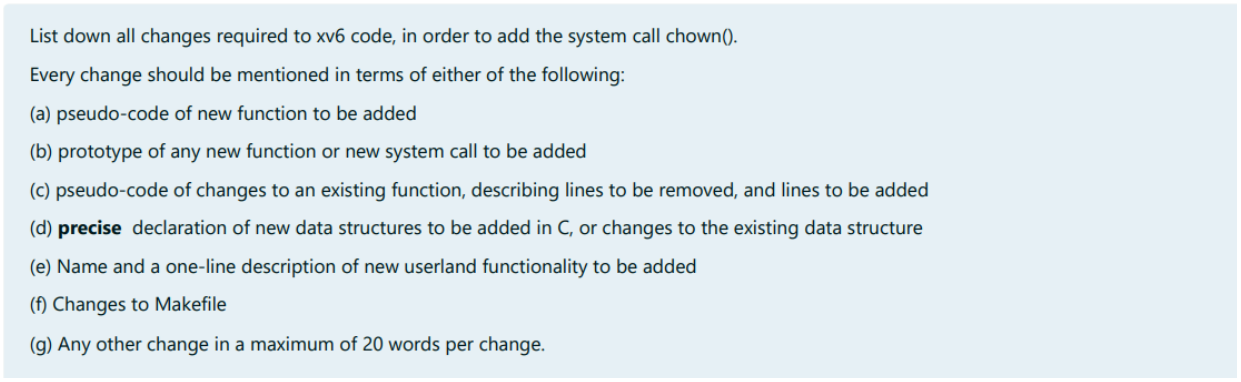
Ans: The computer would have crashed trying to execute the instruction after the one that enabled paging. Mostly due to the fact that you will not be setting the flags to indicate that this 0th entry is present, writable & 4Mb in size. Accessing the 0th entry will hence cause a page fault to occur. → Refer mmu.h and main.c

24. Exactly how many page frames are created by knit1?
Ans: According to last T1 attempt, value of the variable ‘end’ passed to kinit1 is: 801154a8= KERNBASE+1135784. P2V(4*1024*1024) is KERNBASE+(4*1024*1024). Hence Diff between vstart and vend is (4*1024*1024)-1135784=3058520. Hence total pages=3058520/PGSIZE = 746 pages. (unless i have to further divide by 8 as this may be in terms of bytes and I used bits…)

25. Draw a diagram of the data structure changes done by mappage(), walkpgdir(), allocuvm(), deallocuvm(), freeuvm(), setupkvm(). Write 3-4 line description of each of these functions.

26. Take an application, for example cat.c ; Examine it's object code file using objdump. Obtain all the addresses uses for code (text), data, bss, etc. Then draw the page table that will be setup for this application, showing exactly the indices in page-directory and page-table that will be used.

27.


List down all changes required to xv6 code, in order to add the system call chown().

Every change should be mentioned in terms of either of the following:

(a) pseudo-code of new function to be added

(b) prototype of any new function or new system call to be added

(c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added

(d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure

(e) Name and a one-line description of new userland functionality to be added

(f) Changes to Makefile

(g) Any other change in a maximum of 20 words per change.

Ans: Added pseudo code of system call in sysfile.c

```
int
sys_chown(void)
{
  //takes two parameters path and owner
  char *path;
  int owner;
  // checking parameter was given or not and retuen to chown() code
  return chown(path, owner);
}
```

MORE Specific:

```
int sys_chown(void)
{
    char* path;
    int uid, gid;

    if(argstr(0, &path) < 0 || argint(1, &uid)<0 || argint(2, &gid)<0)
            return -1;

    struct proc* currProc=myproc();
    struct inode* ip=namei(path);
    if(currProc->uid!=ROOT)
            return -1;

    chown(uid, gid, ip);
    return 1;
}

void chown(int uid, int gid, struct inode* ip)
{
    ilock(ip);
    ip->uid=uid; —> need some changed in struct INode
    ip->gid=gid;
    iunlock(ip);
}
```

So add uid and gid fields in the struct inode too.

Add a prototype for the new system call to the file syscall.h
int chown(char *path, int uid, int gid);

add implementation in fs.c

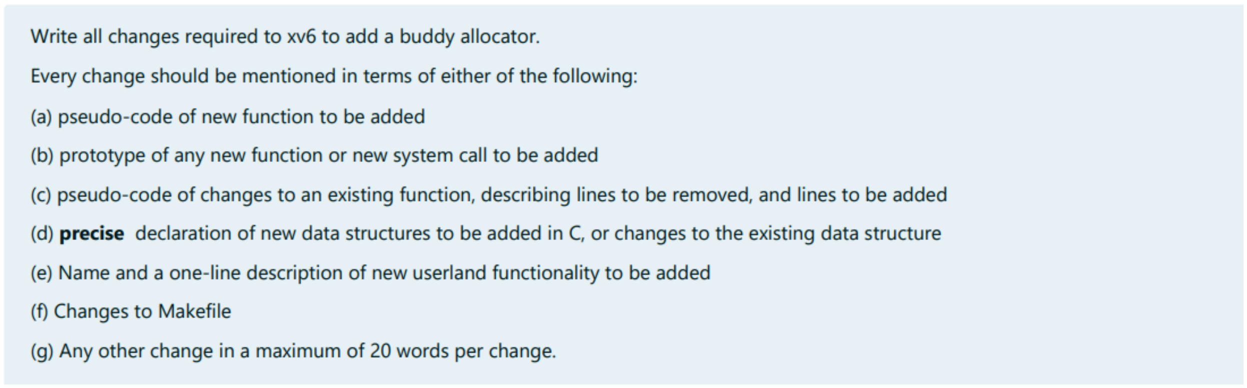
Changes to old functions:
1. Iupdate : set appropriate uid and gid of dip
2. Set uid of root user as 0
3. Every proc also must have a user id (uid)
4. In userinit by default set the UID as ROOT uid
5. In fork() make sure new proc uid= old parent proc uid

Changing system call number in syscall.h by adding the following line:
#define SYS_chown 22

Update system call jump table in syscall.c :
[SYS_chown] sys_chown,
\
Add a test case for the new system call to the file usertests.c. The test case should verify that chown() changes the owner of a file successfully.

Modify the Makefile to include the new source files created for the new system call:
_\chown

28.



Write all changes required to xv6 to add a buddy allocator.

Every change should be mentioned in terms of either of the following:

(a) pseudo-code of new function to be added

(b) prototype of any new function or new system call to be added

(c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added

(d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure

(e) Name and a one-line description of new userland functionality to be added

(f) Changes to Makefile

(g) Any other change in a maximum of 20 words per change.

29. Signaling concept in OS

In operating systems, signaling refers to the mechanism through which processes communicate with each other or with the kernel. It's a crucial aspect for synchronization, inter-process communication (IPC), and handling various events. Here's a breakdown of its key aspects:

1. Process Communication: Processes often need to communicate with each other, whether to coordinate actions, share data, or

notify about events. Signaling provides a way for one process to notify another process or the operating system kernel about a particular event or condition.
2. Synchronization: Signaling is used for synchronization purposes, ensuring that certain events happen in a coordinated manner. For example, a process might signal another process when it finishes its task, allowing the second process to proceed.
3. Handling Events: Operating systems use signals to handle various events such as errors, interrupts, or exceptional conditions. For instance, a process might receive a signal indicating that it attempted an illegal operation, prompting the operating system to terminate the process.
4. Signal Types: Signals can be categorized into different types based on their origin and purpose. Some common types include:
   4.1 Synchronous Signals: Generated by the kernel in response to events such as illegal instruction or divide by zero.
   4.2 Asynchronous Signals: Sent from one process to another or from the kernel to a process to indicate an event or a condition.
   4.3 Software Signals: Generated by the process itself, often using system calls like kill() in Unix-like systems.
   4.4 Hardware Signals: Generated by hardware devices to signal the CPU about events such as interrupts.
5. Signal Handling: Processes can define how they handle signals. They can choose to ignore signals, handle them with custom signal handlers, or let the default action take place. Signal handlers are functions defined by processes to execute specific actions when a signal is received.
6. Examples: Common examples of signals include SIGINT (generated by pressing Ctrl+C in Unix-like systems), which typically terminates the process, and SIGSEGV (segmentation fault), which indicates a memory access violation.

Overall, signaling is a fundamental concept in operating systems, enabling processes to communicate, synchronize, and handle various events efficiently.

30. Threads
->

Threads in operating systems are lightweight processes within a process. Unlike traditional processes, which are independent execution units with separate memory spaces, threads within a process share the same memory space and resources, such as file descriptors and signals. Here's a breakdown of key points regarding threads in operating systems:

Thread Creation: Threads are created within a process using system calls or thread libraries provided by the operating system. Common thread creation methods include pthread_create() in POSIX-compliant systems (like Unix/Linux) and CreateThread() in Windows.
Thread Execution: Threads within a process share the same memory space, allowing them to access shared data and resources directly. Each thread has its own stack for maintaining local variables and function call information. However, they share the heap and global variables.
Concurrency: Threads execute concurrently within a process, meaning they can run simultaneously on multiple CPU cores if available. This concurrency allows for parallelism and efficient utilization of multicore systems.
Thread Synchronization: Since threads share data and resources within a process, synchronization mechanisms are necessary to ensure data consistency and prevent race conditions. Techniques such as mutexes, semaphores, and condition variables are commonly used for thread synchronization.
Thread Communication: Threads within a process can communicate with each other through shared memory or synchronization primitives. This communication enables cooperation and coordination between threads working on different parts of a task.
Thread States: Threads typically have states such as running, ready, blocked, or terminated. The operating system scheduler determines which thread to execute based on its state and priority.
Benefits of Threads: Threads offer several advantages, including:
Reduced overhead compared to processes since they share resources within the same address space.
Improved responsiveness and throughput by utilizing multiple CPU cores efficiently.
Simplified communication and coordination within a process, leading to more streamlined program design.
Challenges: Despite their benefits, threads also introduce challenges such as:
Increased complexity due to shared resources and potential for race conditions.
Difficulty in debugging and reasoning about concurrent programs.
Potential for deadlock and livelock situations if synchronization is not handled properly.
Overall, threads play a crucial role in modern operating systems, enabling concurrent execution and efficient resource utilization within

processes. However, developers need to carefully manage synchronization and communication between threads to ensure correct and reliable behavior.