Lecture Notes on Operating Systems

Problem Set: xv6

1. Consider the following lines of code in a program running on xv6.

```
int ret = fork();
if(ret==0) { //do something in child}
else { //do something in parent}
```

   (a) When a new child process is created as part of handling fork, what does the kernel stack of the new child process contain, after fork finishes creating it, but just before the CPU switches away from the parent?

   (b) How is the kernel stack of the newly created child process different from that of the parent?

   (c) The EIP value that is present in the trap frames of the parent and child processes decides where both the processes resume execution in user mode. Do both the EIP pointers in the parent and child contain the same logical address? Do they point to the same physical address in memory (after address translation by page tables)? Explain.

   (d) How would your answer to (c) above change if xv6 implemented copy-on-write during fork?

   (e) When the child process is scheduled for the first time, where does it start execution in kernel mode? List the steps until it finally gets to executing the instruction after fork in the program above in user mode.

Ans:

   (a) It contains a trap frame, followed by the context structure.

   (b) The parent's kernel stack has only a trap frame, since it is still running and has not been context switched out. Further, the value of the EAX register in the two trap frames is different, to return different values in parent and child.

   (c) The EIP value points to the same logical address, but to different physical addresses, as the parent and child have different memory images.

   (d) With copy-on-write, the physical addresses may be the same as well, as long as both parent and child have not modified anything.

   (e) Starts at forkret, followed by trapret. Pops the trapframe and starts executing at the instruction right after fork.

2. Suppose a machine (architecture: x86, single core) has two runnable processes P1 and P2. P1 executes a line of code to read 1KB of data from an open file on disk to a buffer in its memory. The content requested is not available in the disk buffer cache and must be fetched from disk. Describe what happens from the time the instruction to read data is started in P1, to the time it completes (causing the process to move on to the next instruction in the program), by answering the following questions.

   (a) The code to read data from disk will result in a system call, and will cause the x86 CPU to execute the `int` instruction. Briefly describe what the CPU's `int` instruction does.

   (b) The `int` instruction will then call the kernel's code to handle the system call. Briefly describe the actions executed by the OS interrupt/trap/system call handling code before the read system call causes P1 to block.

   (c) Now, because process P1 has made a blocking system call, the CPU scheduler context switches to some other process, say P2. Now, the data from the disk that unblocks P1 is ready, and the disk controller raises an interrupt while P2 is running. Describe the actions performed by P2 in kernel mode when servicing this disk interrupt.

   (d) Right after the disk interrupt handler has successfully serviced the interrupt above, and before any further calls to the scheduler to context switch from P2, what is the state of process P1?

   Ans:

   (a) The CPU switches to kernel mode, switches to the kernel stack of the process, and pushes some registers like the EIP onto the kernel stack.

   (b) The kernel pushes a few other registers, updates segment registers, and starts executing the system call code, which eventually causes P1 to block.

   (c) P2 saves user context, switches to kernel mode, services the disk interrupt that unblocks P1, and resumes its execution.

   (d) Ready / runnable.

3. Modern operating systems disable interrupts on specific cores when they need to turn off preemption, e.g., when holding a spin lock. For example, in xv6, interrupts can be disabled by a function call `cli()`, and reenabled with a function call `sti()`. However, functions that need to disable and enable interrupts do not directly call the `cli()` and `sti()` functions. Instead, the xv6 kernel disables interrupts (e.g., while acquiring a spin lock) by calling the function `pushcli()`. This function calls `cli()`, but also maintains of count of how many push calls have been made so far. Code that wishes to enable interrupts (e.g., when releasing a spin lock) calls `popcli()`. This function decrements the above push count, and enables interrupts using `sti()` only after the count has reached zero. That is, it would take two calls to `popcli()` to undo the effect of two `pushcli()` calls and restore interrupts. Provide one reason why modern operating systems use this method to disable/enable interrupts, instead of directly calling the `cli()` and `sti()` functions. In other words, explain what would go wrong if every call

to `pushcli()` and `popcli()` in xv6 were to be replaced by calls to `cli()` and `sti()` respectively.

Ans: If one acquires multiple spinlocks (say, while serving nested interrupts, or for some other reason), interrupts should be enabled only after locks have been released. Therefore, the push and pop operations capture how many times interrupts have been disabled, so that interrupts can be reenabled only after all such operations have been completed.

4. Consider an operating system where the list of process control blocks is stored as a linked list sorted by pid. The implementation of the wakeup function (to wake up a process waiting on a condition) looks over the list of processes in order (starting from the lowest pid), and wakes up the first process that it finds to be waiting on the condition. Does this method of waking up a sleeping process guarantee bounded wait time for every sleeping process? If yes, explain why. If not, describe how you would modify the implementation of the wakeup function to guarantee bounded wait.

   Ans: No, this design can have starvation. To fix it, keep a pointer to where the wakeup function stopped last time, and continue from there on the next call to wakeup.

5. Consider an operating system that does not provide the `wait` system call for parent processes to reap dead children. In such an operating system, describe one possible way in which the memory allocated to a terminated process can be reclaimed correctly. That is, identify one possible place in the kernel where you would put the code to reclaim the memory.

   Ans: One possible place is the scheduler code itself: while going over the list of processes, it can identify and clean up zombies. Note that the cleanup cannot happen in the exit code itself, as the process memory must be around till it invokes the scheduler.

6. Consider a process that invokes the `sleep` function in xv6. The process calling sleep provides a lock `lk` as an argument, which is the lock used by the process to protect the atomicity of its call to sleep. Any process that wishes to call `wakeup` will also acquire this lock `lk`, thus avoiding a call to wakeup executing concurrently with the call to sleep. Assume that this lock `lk` is not `ptable.lock`. Now, if you recall the implementation of the sleep function, the lock `lk` is released before the process invokes the scheduler to relinquish the CPU. Given this fact, explain what prevents another process from running the wakeup function, while the first process is still executing sleep, after it has given up the lock `lk` but before its call to the scheduler, thus breaking the atomicity of the sleep operation. In other words, explain why this design of xv6 that releases `lk` before giving up the CPU is still correct.

   Ans: Sleep continues to hold ptable.lock even after releasing the lock it was given. And wakeup requires ptable.lock. Therefore, wakeup cannot execute concurrently with sleep.

7. Consider the `yield` function in xv6, that is called by the process that wishes to give up the CPU after a timer interrupt. The yield function first locks the global lock protecting the process table (`ptable.lock`), before marking itself as RUNNABLE and invoking the scheduler. Describe what would go wrong if `yield` locked `ptable.lock` AFTER setting its state to RUNNABLE, but before giving up the CPU.

Ans: If marked runnable, another CPU could find this process runnable and start executing it. One process cannot run on two cores in parallel.

8. Provide one reason why a newly created process in xv6, running for the first time, starts its execution in the function `forkret`, and not in the function `trapret`, given that the function `forkret` almost immediately returns to `trapret`. In other words, explain the most important thing a newly created process must do before it pops the trap frame and executes the return from the trap in `trapret`.

   Ans: It releases ptable.lock and preserves the atomicity of the context switch.

9. Consider the context switch of a CPU from the context of process P1 to that of process P2 in xv6. Consider the following two events in the chronological order of the events during the context switch: (E1) the ESP (stack pointer) shifts from pointing to the kernel stack of P1 to the kernel stack of P2; (E2) the EIP (program counter) shifts from pointing to an address in the memory allocated to P1 to an address in the memory allocated to P2. Which of the following statements is/are true regarding the relative ordering of events E1 and E2?

   A. E1 occurs before E2.

   B. E2 occurs before E1.

   C. E1 and E2 occur simultaneously via an atomic hardware instruction.

   D. The relative ordering of E1 and E2 can vary from one context switch to the other.

   Ans: A

10. Consider a process P in xv6 that acquires a spinlock L, and then calls the function `sleep`, providing the lock L as an argument to `sleep`. Under which condition(s) will lock L be released *before* P gives up the CPU and blocks?

    A. Only if L is `ptable.lock`

    B. Only if L is not `ptable.lock`

    C. Never

    D. Always

    Ans: B

11. Consider a system running the xv6 OS. The shell process P asks the user for a command, and forks a child process C. C then runs `exec` to execute the `ls` command typed by the user. Assume that process C has just returned from the `exec` system call into user space, and is ready to execute the first instruction of the `ls` binary. Now, the memory belonging to process C would have been allocated and modified at various points during the execution of the `fork` and `exec` system calls. For each piece of memory that belongs to process C that is listed below, you must answer *when* the memory allocation/initialization described in the question occurred. Your possible choices for the answers are given below, and you must pick one of the choices.

    - **Fork+allocproc**: In the `allocproc` function called from `fork` by the parent P.
    - **Fork+...**: In a function (you must provide the name) called from `fork` by the parent P.
    - **Fork**: In the `fork` system call execution by parent P.
    - **Exec+allocuvm**: In the `allocuvm` function called by `exec` in child C.
    - **Exec+loaduvm**: In the `loaduvm` function called by `exec` in child C.
    - **Exec+walkpgdir**: In the `walkpgdir` function called by `exec` in child C.
    - **Exec+...**: In a function (you must provide the name) called from `exec` in child C.
    - **Exec**: In the `exec` system call execution by C.
    - **Other**: Write down any other answer that you feel is correct but is not listed above.

    Note that when a more specific choice is the correct answer, less specific choices will only get partial credit (e.g., if the correct answer is "fork+allocproc", the answer "fork" will only get partial credit).

    (a) When was the the `struct proc` object of C assigned to C from an unused state?

    (b) When was the memory page that holds C's kernel stack assigned to process C from the list of free pages?

    (c) When were the memory pages that hold C's *current* page table entries allocated for this purpose from the list of free pages?

    (d) When were the memory pages that hold the code of the `ls` executable allocated to C from the list of free pages?

    (e) When were the memory pages that hold the code of C's `ls` executable populated with the `ls` binary content from the disk?

    Ans:

    (a) fork+allocproc

    (b) fork+allocproc

    (c) exec+walkpgdir

    (d) exec+allocvm

    (e) exec+loaduvm

12. Consider a system with V bytes of virtual address space available per process, running an xv6-like OS. Much like with xv6, low virtual addresses, up to virtual address U, hold user data. The kernel is mapped into the high virtual address space of every process, starting at address U and upto the maximum V. The system has P bytes of physical memory that must all be usable. The first K bytes of the physical memory holds the kernel code/data, and the rest $P - K$ bytes are free pages. The free pages are mapped once into the kernel address space, and once into the user part of the address space of the process they are assigned to. Like in xv6, the kernel maintains page table mappings for the free pages even after they have been assigned to user processes. The OS does not use demand paging, or any form of page sharing between user space processes. The system must be allowed to run up to N processes concurrently.

    (a) Assume $N = 1$. Assume that the values of V, U, and K are known for a system. What values of P (in terms of V, U, K) will ensure that all the physical memory is usable?

    (b) Assume the values of V, K, and N are known for a system, but the value of P is not known apriori. Suggest how you would pick a suitable value (or range of values) for U. That is, explain how the system designer must split the virtual address space into user and kernel parts.

    Ans:

    (a) The kernel part of the virtual address space of a process should be large enough to map all physical memory, so $V - U \geq P$. Further, the user part of the virtual address space of a process should fit within the free physical memory that is left after placing the kernel code, so $U \leq P - K$. Putting these two equations together will get you a range for P.

    (b) If there are N processes, the second equation above should be modified so that the combined user part of the N processes can fit into the free physical pages. So we will have $N * U \leq P - K$. We also have $P \leq V - U$ as before. Eliminating P (unknown), we get $U \leq \frac{V-K}{N+1}$.

13. Consider a system running the xv6 OS. A parent process P has forked a child C, after which C executes the `exec` system call to load a different binary onto its memory image. During the execution of `exec`, does the kernel stack of C get reinitialized or reallocated (much like the page tables of C)? If it does, explain what part of `exec` performs the reinitialization. If not, explain why not.

    Ans: The kernel stack cannot be reallocated during `exec`, because the kernel code is executing on the kernel stack itself, and releasing the stack on which the kernel is running would be disastrous. Small changes are made to the trap frame however, to point to the start of the new executable.

14. The xv6 operating system does not implement copy-on-write during fork. That is, the parent's user memory pages are all cloned for the child right at the beginning of the child's creation. If xv6 were to implement copy-on-write, briefly explain how you would implement it, and what changes need to be made to the xv6 kernel. Your answer should not just describe what copy-on-write is (do not say things like "copy memory only when parent or child modify it"), but

instead concretely explain *how* you would ensure that a memory page is copied only when the parent/child wishes to modify it.

Ans: The memory pages shared by parent and child would be marked read-only in the page table. Any attempt to write to the memory by the parent or child would trap the OS, at which point a copy of the page can be made.

15. Consider a process P in xv6 that has executed the `kill` system call to terminate a victim process V. If you recall the implementation of `kill` in xv6, you will see that V is not terminated immediately, nor is its memory reclaimed during the execution of the `kill` system call itself.

    (a) Give one reason why V's memory is not reclaimed during the execution of `kill` by P.

    (b) Describe when V is actually terminated by the kernel.

    Ans: Memory cannot be reclaimed during the kill itself, because the victim process may actually be executing on another core. Processes are periodically checked for whether they have been killed (say, when they enter/exit kernel mode), and termination and memory reclamation happens at a time that is convenient to the kernel.

16. Consider the implementation of the `exec` system call in xv6. The implementation of the system call first allocates a new set of page tables to point to the new memory image, and switches page tables only towards the end of the system call. Explain why the implementation keeps the old page tables intact until the end of `exec`, and not rewrite the old page tables directly while building the new memory image.

    Ans: The `exec` system call retains the old page tables, so that it can switch back to the old image and print an error message if exec does not succeed. If exec succeeds however, the old memory image will no longer be needed, hence the old page tables are switched and freed.

17. Consider the list of free pages populated by the xv6 kernel during bootup, as part of the functions `kinit1` and `kinit2`. Which of the following is/are potentially stored in these free pages subsequently, during the running of the system?

    A. Page table mappings of kernel memory pages.

    B. Page table mappings of user memory pages.

    C. The kernel bootloader.

    D. User executables and data.

    Ans: ABD

18. Consider the logical addresses assigned to various parts of code in the kernel executable of xv6. Which of the following statements is/are true regarding the values of the logical addresses?

    A. All are low addresses starting at 0.

    B. All are high address starting at a point after user code in the virtual address space.

    C. Some parts of the kernel code run at low addresses while the rest use high addresses.

D. The answer is dependent on the number of CPU cores.

Ans: C

19. Consider the following `struct pipe` declaration in xv6.

```
#define PIPESIZE 512

struct pipe {
  struct spinlock lock;
  char data[PIPESIZE];
  uint nread;     // number of bytes read
  uint nwrite;    // number of bytes written
  int readopen;   // read fd is still open
  int writeopen;  // write fd is still open
};
```

Below are the definition of the functions invoked when a process reads from or writes into a pipe. You must write C-like code to fill in the core logic of these functions. You may ignore checks for a process being killed, checks for permissions, and other things that are orthogonal to the producer-consumer logic of the pipe, but you must accurately capture the data transfer to/from the pipe buffer, as well as any waiting/signaling involved.

```
int piperead(struct pipe *p, char *addr, int n);
int pipewrite(struct pipe *p, char *addr, int n);
```

In the parameters passed to these functions above, the `addr` variable is a char buffer of size `n`. In case of a pipe read, you must copy data from the pipe buffer into this buffer; in the case of a write, you must copy data out of `addr` into the pipe buffer. You can use the xv6 functions `acquire` and `release` to acquire and release a spinlock. You can also invoke the xv6 functions `sleep` and `wakeup` for conditional wait and signaling. You may recall that the `sleep` function takes a channel and a spinlock as arguments, while the `wakeup` functions take a channel as an argument.

Ans: Pipe read/write logic (simplified version of xv6 code)

```
int
piperead(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  while(p->nread == p->nwrite)
      sleep(&p->nread, &p->lock); //nothing to read
```

```
  for(i = 0; i < n; i++){
    if(p->nread == p->nwrite)
      break; //done reading all that was written, return i
    addr[i] = p->data[p->nread++ % PIPESIZE];
  }
  wakeup(&p->nwrite);
  release(&p->lock);
  return i;
}

int
pipewrite(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  for(i = 0; i < n; i++){
    while(p->nwrite == p->nread + PIPESIZE){
       wakeup(&p->nread);
       sleep(&p->nwrite, &p->lock);
    }
    p->data[p->nwrite++ % PIPESIZE] = addr[i];
  }
  wakeup(&p->nread);
  release(&p->lock);
  return n;
}
```

20. Consider two active processes in xv6 that are connected by a pipe. Process W is writing to the pipe continuously, and process R is reading from the pipe. While these processes are alternately running on the single CPU of the machine, no other user process is scheduled on the CPU. Also assume that these processes always give up CPU due to blocking on a full/empty pipe buffer rather than due to yielding on a timer interrupt . List the sequence of events that occur from the time W starts execution, to the time the context is switched to R, to the time it comes back to W again. You must list all calls to the functions `sleep` and `wakeup`, and calls to `sched` to give up CPU; clearly state which process makes these calls from which function. Further, you must also list all instances of acquiring and releasing `ptable.lock`. Make your description of the execution sequence as clear and concise as possible.

Ans:

(a) W calls `wakeup` to mark R as ready.

(b) W realizes the pipe buffer is full and calls `sleep`.

(c) W acquires `ptable.lock`.

(d) W gives up the CPU in `sched`.

(e) The OS performs a context switch from W to R and R starts execution.

(f) R releases `ptable.lock`.

(g) R calls `wakeup` to mark W as ready.

(h) R realizes the pipe buffer is full and calls `sleep`.

(i) R gives up the CPU in `sched`.

(j) The OS performs a context switch from R to W and W starts execution

21. Consider a newly created process in xv6. Below are the several points in the code that the new process passes through before it ends up executing the line just after the fork statement in its user space. The EIP of each of these code locations exists on the kernel stack when the process is scheduled for the first time. For each of these locations below, precisely explain where on the kernel stack the corresponding EIP is stored (in which data structure etc.), and when (as part of which function or stage of process creation) is the EIP written there. Be as specific as possible in your answers. Vague answers like "during process creation" will not get credit.

    (a) `forkret`

    (b) `trapret`

    (c) just after the `fork()` statement in userspace

    Ans:

    (a) EIP of forkret is stored in struct context by allocproc.

    (b) EIP of trapret is stored on kernel stack by allocproc.

    (c) EIP of fork system call code is stored in trapframe in parent, and copied to child's kernel stack in the fork function.

22. Consider a system running xv6. You are told that a process in kernel mode acquires a spinlock (it can be any of the locks in the kernel code, you are not told which one). While the process holds this spin lock, is it correct OS design for it to:

    (a) process interrupts on the core in which it is running?

    (b) call the `sched` function to give up the CPU?

    For each question above, you must first answer Yes or No. If your answer is yes, you must give an example from the code (specify the name of the lock, and any other information about the scenario) where such an event occurs. If your answer is no, explain why such an event cannot occur.

    Ans:

    (a) No, it cannot. The interrupt may also require the same spinlock, leading to a deadlock.

(b) Yes it is possible. Processes giving up CPU call `sched` with `ptable.lock` held.

23. Consider the following snippet of code from the `sleep` function of xv6. Here, `lk` is the lock given to the sleep function as an argument.

```
if(lk != &ptable.lock){
  acquire(&ptable.lock);
  release(lk);
}
```

For each of the snippets of code shown below, explain what would happen if the original code shown above were to be replaced by the code below. Does this break the functionality of `sleep`? If yes, explain what would go wrong. If not, explain why not.

(a)
```
acquire(&ptable.lock);
release(lk);
```

(b)
```
release(lk);
acquire(&ptable.lock);
```

Ans:

(a) This code will deadlock if the lock given to sleep is `ptable.lock` itself.

(b) A wakeup may run between the release and acquire steps, leading to a missed wakeup.

24. In a system running xv6, for every memory access by the CPU, the function `walkpgdir` is invoked to translate the logical address to physical address. [T/F]

Ans: F

25. Consider a process that has performed a blocking disk read, and has been context switched out in xv6. Now, when the disk interrupt occurs with the data requested by the process, the process is unblocked and context switched in immediately, as part of handling the interrupt. [T/F]

Ans: F

26. Consider a process in xv6 that has been context-switched out because of making a blocking `read` system call. The EIP of the user program instruction that made the system call is stored in the `struct context` on the kernel stack. [T/F]

Ans: F

27. In xv6, state the system call(s) that result in new `struct proc` objects being allocated.

Ans: fork

28. Give an example of a scenario in which the xv6 dispatcher / `swtch` function does NOT use a `struct context` created by itself previously, but instead uses an artificially hand-crafted `struct context`, for the purpose of restoring context during a context switch.

Ans: When running process for first time (say, after fork).

29. Give an example of a scenario in xv6 where a `struct context` is stored on the kernel stack of a process, but this context is never used or restored at any point in the future.

    Ans: When process has finished after exit, its saved context is never restored.

30. Consider a process in xv6 that makes the `exec` system call. The EIP of the `exec` instruction is saved on the kernel stack of the process as part of handling the system call. When and under what conditions is this EIP restored from the stack causing the process to execute the statement after `exec`?

    Ans: If some error occurs during exec, the process uses the eip on trap frame to return to instruction after exec in the old memory image.

31. In xv6, when a process calls `sleep` to block on a disk read, suggest what could be used as a suitable channel argument to the `sleep` function (and subsequently by `wakeup`), in order for the sleep and wakeup to happen correctly.

    Ans: Address of struct buf (can be block number also?)

32. In xv6, when a process calls `wait` to block for a dead child, suggest what could be used as a suitable channel argument in the `sleep` function (and subsequently by `wakeup`), in order for the sleep and wakeup to happen correctly.

    Ans: Address of parent struct proc (can be PID of parent?)

33. State one advantage of the disk buffer cache layer in xv6 besides caching. Put another way, even if there was zero cache locality, the higher layers of the xv6 file system would still have to use the buffer cache: state one functionality of the disk buffer cache (besides caching) that is crucial for the higher layers.

    Ans: Synchronization - only one process at a time handles a disk block.

34. Consider a parent process P that has executed a fork system call to spawn a child process C. Suppose that P has just finished executing the system call code, but has not yet returned to user mode. Also assume that the scheduler is still executing P and has not context switched it out. Below are listed several pieces of state pertaining to a process in xv6. For each item below, answer if the state is identical in both processes P and C. Answer Yes (if identical) or No (if different) for each question.

    (a) Contents of the PCB (struct proc). That is, are the PCBs of P and C identical? (Yes/No)

    (b) Contents of the memory image (code, data, heap, user stack etc.).

    (c) Contents of the page table stored in the PCB.

    (d) Contents of the kernel stack.

    (e) EIP value in the trap frame.

    (f) EAX register value in the trap frame.

    (g) The physical memory address corresponding to the EIP in the trap frame.

(h) The files pointed at by the file descriptor table. That is, are the file structures pointed at by any given file descriptor identical in both P and C?

Ans:

(a) No

(b) Yes

(c) No

(d) No

(e) Yes

(f) No

(g) No

(h) Yes

35. Suppose the kernel has just created the first user space "init" process, but has not yet scheduled it. Answer the following questions.

(a) What does the EIP in the trap frame on the kernel stack of the process point to?

(b) What does the EIP in the context structure on the kernel stack (that is popped when the process is context switched in) point to?

Ans:

(a) address 0 (first line of code in init user code)

(b) forkret / trapret

36. Consider the exit system call in xv6. The exit function acquires ptable.lock before giving up the CPU (in the function sched) for one last time. Who releases this lock subsequently?

Ans: The process that runs immediately afterwards (or scheduler)

37. Consider the following actions that happen during a context switch from thread/process P1 to thread/process P2 in xv6. (One of P1 or P2 could be the scheduler thread as well.) Arrange the actions below in chronological order, from earliest to latest.

(A) Switch ESP from kernel stack of P1 to that of P2

(B) Pop the callee-save registers from the kernel stack of P2

(C) Push the callee-save registers onto the kernel stack of P1

(D) Push the EIP where execution of P1 stops onto the kernel stack of P1.

Ans: DCAB

38. xv6 always performs a context switch by switching to the scheduler thread first, and then switching from the scheduler thread to the context of the next process. However, is it possible for any other OS design to switch directly from the context of one process to the context of another without going via a separate scheduler thread? (Yes/No)

    Ans: Yes

39. In xv6, is it possible that one process acquires a lock, but some other process releases it? If yes, give an example where such a thing happens. If not, explain why not.

    Ans: Yes. ptable.lock during context switch.

40. In xv6, when a process calls wakeup on a channel to wakeup another process, does this lead to an immediate context switch of the process that called wakeup (immediately after the wakeup instruction)? (Yes/No)

    Ans: No

41. When a process terminates in xv6, when is the struct proc entry of the process marked as unused/free?

    A. During the execution of exit B. During the sched function that performs the context switch C. In the scheduler, when it iterates over the array of all struct proc D. During the execution of wait by the parent

    Ans: D

42. Why does the function kinit1 called by the xv6 main() map only free pages within the first 4MB of RAM? Why does it not collect free pages from the entire available RAM?

    Ans: Because it needs some free pages for page tables before it can map the entire memory.

43. Why does the page table created by the entry code (entrypgdir) add a mapping from virtual addresses [0,4MB] to physical addresses [0,4MB]?

    Ans: So that the instructions between turning on MMU and jumping to high address space run correctly.

44. Suppose xv6 is running on a machine with 16GB virtual address space, out of which 0-8GB are reserved for user code, and the remaining addresses are available to the kernel. What is the maximum amount of RAM that xv6 can successfully manage and address in this system? Assume that you can change the values of KERNBASE, PHYSTOP and other such constants in the code suitably.

    Ans: 8GB

45. Does the kernel stack of the process calling exec get reallocated during the execution of the exec system call? Answer yes/no and justify.

    Ans: No, because the kernel is executing on the kernel stack.

46. Do the pages holding the memory image of the process calling exec get reallocated during the execution of the exec system call? Answer yes/no and justify.

    Ans: Yes, because a new memory image would be loaded.

47. Do the pages that hold the page table entries of a process calling exec get reallocated during the execution of the exec system call? Answer yes/no and justify.

    Ans: Yes, to point to the new memory image

48. The function walkpgdir is invoked on every memory access by the CPU, to translate virtual addresses to physical addresses. Answer True/False and justify.

    Ans: No, the translation is done in hardware by MMU.

49. Consider two processes in xv6 that both wish to read a particular disk block, i.e., either process does not intend to modify the data in the block. The first process obtains a pointer to the struct buf using the function "bread", but never causes the buffer to become dirty. Now, if the second process calls "bread" on the same block before the first process calls "brelse", will this second call to "bread" return immediately, or would it block? Briefly describe what xv6 does in this case, and justify the design choice.

    Ans: Second call to bread would block. Buffer cache only allows access to one block at a time, since the buffer cache has no control on how the process may modify the data

50. Consider a process that calls the log_write function in xv6 to log a changed disk block. Does this function block the invoking process (i.e., cause the invoking process to sleep) until the changed block is written to disk? Answer Yes/No.

    Ans: No

51. Repeat the previous question for when a process calls the bwrite function to write a changed buffer cache block to disk. Answer Yes/No.

    Ans: Yes

52. When the buffer cache in xv6 runs out of slots in the cache in the bget function, it looks for a clean LRU block to evict, to make space for the new incoming block. What would break in xv6 if the buffer cache implementation also evicted dirty blocks (by directly writing them to their original location on disk using the bwrite function) to make space for new blocks?

    Ans: All writes must happen via logging

53. (a) Recall that buffer caches of operating systems come in two flavors when it comes to writing dirty blocks from the cache to the secondary storage disk: write through caches and write back caches. Consider the buffer cache implementation in xv6, specifically the `bwrite` function. Is this implementation an example of a write through cache or a write back cache? Explain your answer.

(b) If the xv6 buffer cache implementation changed from one mode to the other, give an example of xv6 code that would break, and describe how you would fix it. In other words, if you answered "write through" to part (a) above, you must explain what would go wrong (and how you would fix it) if xv6 moved to a write back buffer cache implementation. And if you answered "write back" to part (a), explain what would need to change if the buffer cache was modified to be write through instead.

(c) The buffer cache in xv6 maintains all the `struct buf` buffers in a fixed-size array. However, an additional linked list structure is imposed on these buffers. For example, each `struct buf` also has pointers `struct buf *prev` and `struct buf *next`. What additional functions do these pointers serve, given that the buffers can all be accessed via the array anyway?

Ans:

(a) Write through cache

(b) If changed to write back, the logging mechanism would break.

(c) Helps implement LRU eviction

54. Consider a system running xv6. A process has the three standard file descriptors (0,1,2) open and pointing to the console. All the other file descriptors in its `struct proc` file descriptor table are unused. Now the process runs the following snippet of code to open a file (that exists on disk, but has not been opened before), and does a few other things as shown below. Draw a figure showing the file descriptor table of the process, relevant entries in the global open file table, and the in-memory inode structures pointed at by the file table, after each point in the code marked parts (a), (b), and (c) below. Your figures must be clear enough to understand what happens to the kernel data structures right after these lines execute. You must draw three separate figures for parts (a), (b), and (c).

```
int fd;
fd = open("foo.txt", O_RDWR); //part (a)
dup(fd);                      //part (b)
fd = open("foo.txt", O_RDWR); //part (c)
```

Ans:

(a) Open creates new FD, open file table entry, and allocated new inode.

(b) Dup creates new FD to point to same open file table entry.

(c) Next open creates new open file table entry to point to the same inode.

55. Consider the execution of the system call `open` in xv6, to create and open a new file that does not already exist.

(a) Describe all the changes to the disk and memory data structures that happen during the process of creating and opening the file. Write your answer as a bulleted list; each item

of the list must describe one data structure, specify whether it is in disk or memory, and briefly describe the change that is made to this data structure by the end of the execution of the open system call.

(b) Suggest a suitable ordering of the changes above that is most resilient to inconsistencies caused by system crashes. Note that the ordering is not important in xv6 due to the presence of a logging mechanism, but you must suggest an order that makes sense for operating systems without such logging features.

Ans:

(a)
- A free inode on disk is marked as allocated for this file.
- An inode from the in-memory inode cache is allocated to hold data for this new inode number.
- A directory entry is written into the parent directory on disk, to point to the new inode.
- An entry is created in the in-memory open file table to point to the inode in cache.
- An entry is added to the in-memory per-process file descriptor table to point to the open file table entry.

(b) The directory entry should be added after the on-disk inode is marked as free. The memory operations can happen in any order, as the memory data structures will not survive a crash.

56. Consider the operation of adding a (hard) link to an existing file /D1/F1 from another location /D2/F2 in the xv6 OS. That is, the linking process should ensure that accessing /D2/F2 is equivalent to accessing /D1/F1 on the system. Assume that the contents of all directories and files fit within one data block each. Let $i(x)$ denote the block number of the inode of a file/directory, and let $d(x)$ denote the block number of the (only) data block of a file/directory. Let L denote the starting block number of the log. Block L itself holds the log header, while blocks starting $L + 1$ onwards hold data blocks logged to the disk.

Assume that the buffer cache is initially empty, except for the inode and data (directory entries) of the root directory. Assume that no other file system calls are happening concurrently. Assume that a transaction is started at the beginning of the link system call, and commits right after the end of it. Make any other reasonable asusmptions you need to, and list them down.

Now, list and explain all the read/write operations that the disk (not the buffer cache) sees during the execution of this link operation. Write your answer as a bulleted list. Each bullet must specify whether the operation is a read or write, the block number of the disk request, and a brief explanation on why this request to disk happens. Your answer must span the entire time period from the start of the system call to the end of the log commit process.

Ans:

- read i(D1), read d(D1)—this will give us the inode number of F1.
- read i(F1). After reading the inode and bringing it to cache, its link count will be updated. At this point, the inode is only updated in the buffer cache.
- read i(D2), read d(D2)—we check that the new file name F2 does not exist in the directory. After this, the directory contents are updated, and a note is made in the log.

- Now, the log starts committing. This transaction has two modified blocks (the inode of F1 and the directory content of D2). So we will see two disk blocks written to the log: write to L+1 and L+2, followed by a write to block L (the log header).
- Next, the transactions are installed: a write to disk blocks i(F1) and d(D2).
- Finally, another write to block L to clear the transaction header.

57. Which of the following statements is/are true regarding the file descriptor (FD) layer in xv6?

A. The FD returned by `open` is an index to the global `struct ftable`.

B. The FD returned by `open` is an index to the open file table that is part of the `struct proc` of the process invoking `open`.

C. Each entry in the global `struct ftable` can point to either an in-memory inode object or a pipe object, but never to both.

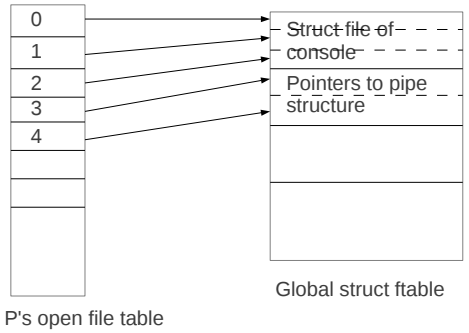
D. The reference count stored in an entry of the `struct ftable` indicates the number of links to the file in the directory tree.

Ans: BC

58. Consider the following snippet of the shell code from xv6 that implements pipes.

```
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
  panic("pipe");
if(fork1() == 0){
close(1);
dup(p[1]);   //part (a)
close(p[0]);
close(p[1]);
runcmd(pcmd->left);
}
if(fork1() == 0){
close(0);
dup(p[0]);
close(p[0]);
close(p[1]);
runcmd(pcmd->right);
}
close(p[0]);
close(p[1]);  //part (b)
wait();
wait();
```

Assume the shell gets the command "echo hello | grep hello". Let P denote the parent shell process that implements this command, and let CL and CR denote the child processes created to execute the left and right commands of the above pipe command respectively. Assume P has no other file descriptors open, except the standard input/output/error pointing to the console. Below are shown the various (global and per-process) open file tables right after P executes the `pipe` system call, but before it forks CL and CR.

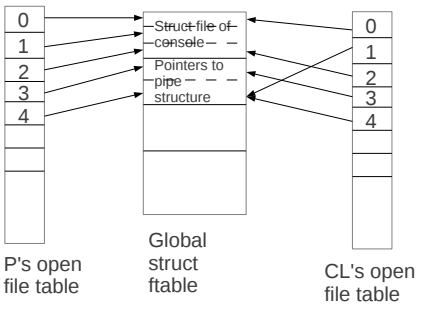


Draw similar figures showing the states of the various open file tables after the execution of lines marked (a) and (b) above. For each of parts (a) and (b), you must clearly draw both the

global and per-process file tables, and illustrate the various pointers clearly. You may assume that all created child processes are still running by the time the line marked part (b) above is executed. You may also assume that the scheduler switches to the child right after fork, so that the line marked part (a) in the child runs before the line marked part (b) in the parent.

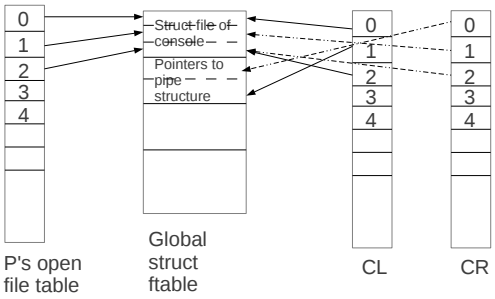
Ans:

(a) The fd 1 of the child process has been modified to point to one of the pipe's file descriptors



by the close and dup operations.

(b) CL and CR are now connected to either ends of the pipe. The parent has no pointers to the



pipe any more.