CS 219 Spring 2025 End-semester examination

(20 questions, 80 marks, 40% weightage)

Name:_____ **Roll number:**_____

1. Consider two processes P1 and P2 in a single core system running xv6. Process P1 makes a blocking disk read system call, and a context switch happens to process P2 via the scheduler thread. Later on, when the DMA-enabled disk completes reading, P2 is interrupted by the disk, goes to kernel mode to handle the interrupt (and mark P1 as ready), and resumes back in user mode again. Below are listed several events that may happen from the time P1 makes the system call to start the disk read to the time when P2 handles the disk interrupt and resumes execution.

   E1. The CPU runs the trap (`int n`) instruction

   E2. The CPU runs the return from trap (`iret`) instruction

   E3. The disk device driver issues a command to read from the disk

   E4. The disk raises an interrupt to the CPU upon completion of the read

   E5. The disk performs DMA of the disk data into the disk buffer cache

   E6. P1 invokes the `sleep` function to relinquish the CPU

   E7. The scheduler thread switches to process P2

   E8. The device driver handles the interrupt from disk and marks P1 as ready to run in the ptable

   List the set of events that occur in chronological order, from earliest to latest, in the time interval between t_1 and t_2, where t_1 and t_2 are defined in the questions below. Consider all events from the above set that can occur in this interval.

   (a) t_1 = the time when P1 makes a blocking system call, t_2 = the time when P2 begins to run in user mode after the context switch
   Ans: E1, E3, E6, E7, E2

   (b) t_1 = the time when the disk hardware completes the reading process, t_2 = the time when P2 resumes execution again in user mode after handling the disk interrupt
   Ans: E5, E4, E1, E8, E2

2. Consider a process in xv6 which is about to be switched out of the CPU. This process can call the `sched` function (which in turn calls `swtch` to switch to the scheduler thread) from three different functions in xv6. List the three functions (in any order) in the three sub-parts given below. For each function, give an example of an event or scenario which causes the process to invoke this function and initiate a context switch via this function.

   (a) Function 1 and when it is invoked:
   Ans: Sleep, when process makes a blocking system call

   (b) Function 2 and when it is invoked:
   Ans: Yield, when process has to switch out due to timer interrupt

(c) Function 3 and when it is invoked:

Ans: Exit, when process calls exit system call

(d) Now, when the scheduler thread switches to a process, it resumes execution in `sched`, which can return to one of 3 different functions. Name the three functions where a process can resume execution after a context switch.

Ans: sleep, yield, forkret

3. Draw the kernel stack of a process P in xv6 in the following scenarios. You must draw the stack vertically, with the top of the stack shown above the bottom. You must clearly label the following structures on the stack if they are present: `trap frame`, `struct context`. For each structure present on the kernel stack, you must also indicate which address the EIP points to.

(a) P is a process that went into kernel mode for a non-blocking system call, and is currently running the system call code in kernel mode

Ans: We expect you to draw a trapframe on kernel stack, where EIP points to userspace code at which trap occurred

(b) P is a newly forked process in xv6 that is waiting for its turn to execute for the first time

Ans: We expect you to draw a trapframe at bottom of stack (EIP points to userspace code at fork), and struct context at top of stack (EIP points to forkret)

4. Consider the page table entry of a process in a Linux-like operating system running on x86 architecture. Answer the following questions in one sentence each.

(a) Suppose this page table entry has both valid and present bits set to true. What happens at the TLB when a virtual address in this page is accessed?

Ans: Can be a TLB hit or miss, cannot say

(b) For the previous question, can this memory access ever result in a trap to the OS? Answer yes/no with justification.

Ans: Yes, can result in a trap if it is a illegal access, e.g., insufficient permissions

(c) Now suppose this page table entry has the valid bit set, but the present bit is not set. What happens at the TLB when a virtual address in this page is accessed?

Ans: Always a TLB miss, since no physical address is present

(d) For the previous question, can this memory access ever result in a trap to the OS? Answer yes/no with justification.

Ans: Yes, always results in a trap

5. Consider an OS that implements an approximate LRU policy in the following manner. Every page table entry has an "accessed" bit which is set by the MMU when the page table entry is accessed. Every 10 milliseconds, the OS inspects these bits for all pages, archives this information, and resets these bits in the page table entries, so that they may be set afresh in the next round. The OS uses the history of the accessed bit for each page over the past 5 rounds (i.e., over the past 50 milliseconds) to pick the least recently used page. If the OS has a choice of multiple pages that appear as LRU by this method, ties are broken arbitrarily.

(a) Now, when the OS needs a free page and is looking for a victim page to evict, it finds there is only one page which appear as LRU by the above algorithm. Is this page guaranteed to be the true LRU page according to an ideal "exact" (not approximate) LRU algorithm that has visibility into all page accesses? Answer yes/no and provide a justification.

Ans: Yes, if only one page is identified as LRU by this algorithm, then this has to be the true LRU page. There is no way the true LRU page will have accessed bit set in a later round as compared to this page identified by the approximate LRU algorithm.

(b) Now, suppose we modify the page replacement policy to make it a Least Frequently Used (LFU) policy. The OS uses information of the accessed bits over the past 50 milliseconds (which are inspected and reset every 10 milliseconds as described above) to count the number of times the accessed bit is set for each page across the past 5 rounds. This information is then used to pick the least frequently used page. At some point, when trying to find a victim page, the algorithm finds that one page has its accessed bit set in only one of the rounds (while all other pages have their accessed bits set in 2 or more rounds) and hence appears as LFU by the algorithm above. Is this page guaranteed to be the true LFU page according to an ideal "exact" (not approximate) LFU algorithm that has visibility into all page accesses? Answer yes/no and provide a justification.

Ans: No. Multiple accesses within a 10 millisecond interval show up as just one access when we inspect the accessed bit. So a page that has accessed bit set in only one round (but was accessed, say, 10 times in that interval) may get picked as LFU, while the true LFU page could have been accessed twice in two different epochs, and missed being identified as LFU by this algorithm.

6. Consider a simple 32-bit OS like xv6. The OS designers have to make the decision of where to place the "boundary" between userspace addresses and kernel addresses in the virtual address space of a process. Recall that xv6 places this boundary at 2GB (KERNBASE). Suppose the OS designers wish to place the boundary at K bytes, i.e., addresses in the range 0 to K bytes will hold userspace mappings, while addresses from K to the maximum value V (which is 4GB) will hold kernelspace mappings.

   (a) We require that the kernel address space in every process be large enough to map all usable physical memory of size P bytes into the high virtual addresses between K and V (4GB), so that every process in kernel mode can access all physical memory by mapping every physical address $p \in [0, P-1]$ to virtual address $K + p$. Write an equation in terms of K, V, and P to capture this constraint.

   Ans: $P \leq V - K$

   (b) We also require that the user part of the address space of every process be at least of size U bytes, so that the userspace code, data, stack, heap and other things can fit comfortably. Using this information, and your answer in the previous part, work out the valid range of values for K that is available to the OS designers.

   Ans: We have $K \geq U$ from this part, and $K \leq V - P$ from the previous part, so the range for K is $[U, V - P]$

7. Describe the output of the following multi-threaded programs. You must list all possible outputs to get credit.

(a) A program spawns three threads, where the argument to the thread is the loop counter itself. The relevant code snippet is shown below. What possible outputs can get printed?

```
void* print_thread_message(void* arg) {
  int thread_num = *(int*)arg;
  printf("%d ", thread_num);
}
int main() {
  pthread_t threads[N];
  for (int i = 0; i < 3; i++)
    pthread_create(&threads[i], NULL, print_thread_message, &i);
 ..
}
```

Ans: Since we are passing the address of the loop counter, the loop counter can change before the thread start function accesses it. So any possible values in 0, 1, 2, 3 may be printed by the three threads. However, if we assume that the thread start functions are run in the same order they are invoked, then the values printed will be monotonically increasing, e.g., 0 1 2, 1 1 2, 1 2 2, 2 2 2. Both possible answers will be considered.

(b) A program spawns N threads, and each thread increments a global variable counter by K (`counter = counter + K`) without using any locks. Note that when this increment operation is translated into assembly code, it looks like this: load value of counter into register, increment register by K, store value into counter variable. What are the possible values of the counter after all threads finish their increment concurrently?

Ans: The counter value can be one of K, $2K$, $3K$, ..., NK.

8. Consider two threads T1 and T2 belonging to the same process, created using the `pthreads` library, running on a multicore system. Given below are different scenarios where the threads use synchronization primitives like locks and condition variables. In each scenario, state whether there is a possibility of a deadlock, and justify your answer in one sentence.

(a) T1 is running on core C1, and acquires a pthreads spinlock mutex. While holding this lock, T1 is switched out. T2 starts running on the same core, and tries to acquire the same pthreads mutex.

Ans: No deadlock. T2 will spin for some time, and gets switched out. Eventually, T1 runs and releases the lock, T2 will acquire the lock the next time it runs. This is inefficient but does not cause a deadlock.

(b) T1 is running on core C1, and acquires a pthreads spinlock mutex. While T1 is holding this lock, T2 starts to run on another core C2, and starts to spin for the same spinlock.

Ans: No deadlock. T2 can spin till T1 releases the lock.

(c) Threads T1 and T2 require that T2 runs after T1. So T2 calls wait on a condition variable, with a pthreads spinlock held. When T1 finishes execution, it calls signal on the same condition variable, but without holding the same spinlock used by T2.

Ans: Yes, a deadlock can occur due to a missed wakeup.

(d) T1 is running on core C1, goes to kernel mode, and acquires a kernel spinlock. While T1 is holding this lock, T2 starts to run on another core C2, goes into kernel mode, and starts to spin for the same kernel spinlock.

Ans: No deadlock, T2 spins safely till T1 releases lock.

9. Consider a program that handles 3 types of requests (let us call them P, Q, and R) that arrive at random times. A new thread is created to handle each request. The program batches requests into groups of three (one each of P, Q, R) and processes them together. A thread waits until the corresponding requests of the other types arrive, and three "matched" threads can run together. The threads must be matched whenever possible, and a thread must not wait unnecessarily if the matching threads have already arrived. You must write pseudocode that synchronizes the threads handling the requests, where the synchronization is done using only semaphores. You must write the code for a thread handling request P below, and the code for requests Q and R will be symmetric. You must declare all the semaphores and any other variables (like boolean flags or integer counters) that you may need in your solution, along with their initial values.

   Ans: Variables needed are three semaphores for P, Q, R (initialized to 0), and 3 counters for number of requests of each type that have arrived (all initialized to 0). We will also need a semaphore to work as a lock to protect these counters (initialized to 1). The code for thread P is given below.

```
down(lock)
countP++
if(countP >= 1 && countQ >= 1 && countQ >= 1) {
  //set is complete
  countP--; countQ--; countR--; //consume
  up(semP); up(semQ); up(semR); //wakeup
}
up(lock)
down(semP)
```

10. Consider an application that stores key-value pairs in a hash table. The application receives requests to get (i.e., read) or put (i.e., write) key-value pairs into the hash table. Every request is handled by a separate thread. In order to improve performance, the program allows multiple threads handling get requests to concurrently access the hash table, because a get request only reads the data structure. However, whenever a thread handling a put request needs to access the hash table, it must do so exclusively, without executing concurrently with any other get or put thread. The application mostly receives get requests, and must optimize for serving get requests as quickly as possible. That is, if get requests arrive while other threads serving get requests already have access to the data structure, then the new get requests must be processed as well, even if doing so increases the waiting time for threads with put requests. You are given the code for the put requests below. You must complete the code for get requests. You must use only semaphores in your solution. One semaphore `semput` (initialized to 1) is already given to you as part of the code of the put thread. You can use two other variables in your solution: semaphore `semget` (initialized to 1), and an integer `count` (initialized to 0). You must not use any other variables in your solution.

    Ans:

```
down(semget) //all other gets wait here
count++
if(count ==1)
```

```
  down(semput) //don't coexist with a put, first get waits here
up(semget)
... access to get ....
down(semget)
count--
if(count == 0)
  up(semput)
up(semget)
```

11. Continuing on the previous problem, consider the same key-value pair system, with the following changes. The system no longer handles put requests, i.e., all requests are get requests that can concurrently access the database. However, we want to ensure that there are no more than N get threads accessing the data structure at any time, in order not to impact system performance. Additional threads beyond the N threads must wait until some thread leaves. Complete the code for the get threads below. You must use only the following variables in your solution: a lock `mutex`, a condition variable `getCV`, and a counter `getcount` (initialized to 0).

    Ans:

```
lock(mutex)
getcount++
while(getcount > N) wait(getCV, mutex)
unlock(mutex)
... access database....
lock(mutex)
getcount--
if(getcount < N) signal(semget) // signal broadcast, signal always also possi
unlock(mutex)
```

12. Consider the following pseudocode to implement the read system call in a simple filesystem. The function shown below takes 4 arguments: the inode of the file from which the data must be read, a pointer to the destination buffer into which the bytes must be copied, the offset at which the read must begin, and the number of bytes to be read. The return value of the function must be the number of bytes actually read. Given below is an explanation of the variables and functions used in the code below.

    - The inode pointer structure `struct inode *ip` has various pieces of metadata about the file available, including the variable `size` which indicates the size of the file.
    - File data is stored on a disk with block size of 512 bytes, and you may assume that the offset `int off` given as argument is a multiple of the block size.
    - The function `get_block_num (struct inode *ip, int k)` returns the disk block number of the k-th block of the file whose inode pointer is `ip`.
      For example, `get_block_num(ip, 0)` returns the disk block number of the first block of the file.
    - The function `diskread(N)` returns a pointer to the data of disk block number `N`. This is a pointer to the bytes located in the disk buffer cache. If the block is not in cache, this function waits for data to be read from disk and then returns the pointer to data.

- The local variable `tot` keeps track of the total number of bytes read so far. Note that a read can span multiple disk blocks.
- The local variable `m` keeps track of how many bytes to read from the block in the current iteration of the for loop. Note that we may not always read the entire 512 bytes, e.g., if the remaining bytes left in the file is less than 512.
- The function `memcpy(char *dst, char *src, int size)` copies `size` bytes into `dst` from `src`.

Fill in the blanks in the pseudocode shown below.

```
read(struct inode *ip, char *dst, int off, int n) {
  int tot, m; char *block
  //update n if not enough bytes to read in the file
  if(off + n > ip->size)     _________________ ;

  for(tot=0; tot<n; tot+=m, off+=m, dst+=m) {

    block = diskread(get_block_num(ip, ___________));

    m = min(___________, _____________); //minimum of 2 numbers

    memcpy(________, _________, _________);
  }
  return n;
}
```

Ans:

```
n = ip->size - off
off/512
min(n - tot, 512)
memcpy(dst, block, m)
```

13. Consider a server that is part of a social media platform, which stores user photographs persistently on disk using a simple filesystem, like the one studied in class. Assume that all photographs are of uniform size 1MB. The server stores 1M (2^{20}) such photographs as separate files in the top-level root directory of the filesystem. A directory entry (i.e., the mapping between the filename and inode number in a directory data block) is 4 bytes in the filesystem. The block size on disk is 1KB. Inodes are 256 bytes in size.

    (a) How many data blocks does the top-level root directory need, in order to store the directory entries corresponding to all the photographs?

    Ans: Each block can hold 256 = 2^8 dir entries. Total files = 2^{20}. Total blocks needed to hold all dir entries = $2^{20}/2^8 = 2^{12} = 4096$

    (b) Assume that the inode in the filesystem has 64 direct blocks, a single indirect block, and a double indirect block (which contains block numbers of more single indirect blocks). Each

indirect block can hold 256 file block numbers. How many single indirect block numbers are stored in the double indirect block of the root inode?

Ans: 15. The double indirect block stores 4096-256-64 = 3776 block numbers. These will fit in approximately 3776/265 which is 15 single indirect blocks.

(c) Suppose we want to allocate enough main memory to cache all the data blocks of the root directory and all inodes of all the photographs, so that any photograph can be directly read from disk, without having to resort to additional disk accesses to read metadata. What is the minimum amount of main memory needed to satisfy this requirement? You may ignore the memory consumed to store the root inode.

Ans: Memory needed for root dir data blocks = 4096 * 1KB = 4MB. Memory needed for all inodes = 256 * 2^{20} = 256MB. Total memory is 260MB.

(d) Consider the following alternate storage model for the photographs that allows us to directly read file data blocks, without needing to read additional metadata blocks. All photographs are stored contiguously one after another in one large file in the root directory. We maintain an index (i.e., a hash table) that maps the name of the photograph to its offset within a file. To read a file, we look up its offset from the hash table, find the block number corresponding to offset from the inode of the large file (that is also cached in memory), and directly read the photograph from disk. The hash table consumes 4 bytes for each file entry stored. How much memory is needed in this scheme to ensure that we can directly access the photographs from disk without reading additional metadata blocks? You may ignore the memory consumed to store the root inode and the inode of the one large file.

Ans: 4 bytes for each of the 1M photos = 4MB.

14. Let us continue with the filesystem specifications of the previous question: 1KB disk blocks, 4 byte directory entries, 256 byte inodes with 64 direct + one single indirect block + one double indirect block, 256 block numbers per indirect block. Now this filesystem is being used to store general files, not just photographs. What is the maximum amount of file data (in bytes) that can be stored across all files in a single directory in this filesystem? You may assume that there is enough space on disk for a very large number of data and metadata blocks, so the limits will come from the number of files that can be stored in a directory, and the maximum size of each such file. Make reasonable assumptions in your solution, and show all your calculations properly.

Ans: Each directory inode has a maximum of $2^6 + 2^8 + 2^{16} = 1029 * 2^6$ data blocks, each with 256 directory entries, so the maximum number of files in each directory = N = $1029 * 2^{14}$.

Each file inode has $2^6 + 2^8 + 2^{16} = 1029 * 2^6$ block numbers, so the maximum size of a file = M = $1029 * 2^{16}$ bytes.

So total size of all files = N*M where N and M are as above. This is approximately 2^{50} bytes.

15. Suppose a filesystem user makes the link system call to link an existing file `/a/foo.txt` with a new pathname `/b/bar.txt`, i.e., an extra link is added to the inode of `foo.txt` from directory `b`. Assume that the filename `bar.txt` does not exist in directory `b` before the system call, and the linking succeeds. Assume there is enough space in the directory data blocks of `b` to add the new directory entry.

(a) This link system call will change two different blocks on disk. What are these two blocks and what are the changes to them?

Ans: inode of foo.txt (link count is incremented) and data block of b (new directory entry added mapping filename bar.txt to foo.txt's inode number)

(b) Suppose there is a power failure while executing this system call. Describe two possible inconsistencies in the filesystem that may arise because of the crash during this system call.

Ans: If link count is updated, but the inode is not linked from b, then the inode has an extra link count, and will never be cleaned up. If the directory entry was added but the link count was not updated, then the inode may go away when all other links to it are unlinked, leaving bar.txt to point to a garbage inode.

16. Consider a simple filesystem like the one discussed in class. For each of the scenarios below, draw all the in-memory data structures (file descriptor arrays, open file table, in-memory inodes) and show the pointers and linkages between them clearly.

(a) Process P opens a file twice, by invoking the open system call 2 times. Another process Q also opens the same file. Show all the in-memory data structures after all of these system calls complete successfully.

Ans: 2 entries in P's file descriptor array, one in Q's file descriptor array. These point to 3 different open file table entries, all of which point to the same inode in memory.

(b) Now, process P forks a child C. Show the in-memory structures again now. You may ignore drawing process Q now, since nothing has changed for it.

Ans: The file descriptor array of P is copied to C, so C also has the file open twice. The file descriptor array entries of C point to the same open file table entries as P, which point to the same inode.

17. Consider a web server that receives 1 M requests/second over the network. Some cores in the system are dedicated to running ksoftirq (botton half of the interrupt handler) processing, while some other cores are dedicated to running a multi-threaded user application that processes these requests. Incoming traffic is distributed equally to the cores running the kernel (ksoftirq) processing first, and then distributed equally to the application cores for further processing. You may ignore the time taken for other tasks like top-half interrupt handling or any other work done on the server.

(a) The ksoftirq thread on each core runs in a tight loop, continuously processing requests without blocking in any way, and needs 20 microseconds to process each request. What is the minimum number of cores that must be running ksoftirq threads in order to process all incoming load in a timely manner?

Ans: 20 cores (each core does 50K req/s)

(b) After kernel processing, the incoming requests are distributed equally and processed by a multithreaded web server program running on multiple cores. Each application thread runs for 10 microseconds on the CPU to handle the request, and spends another 990 microseconds waiting for disk I/O for the request. After spending 1 millisecond processing a request in this way, it moves on to the next request. What is the minimum number of application threads that must be run on each core to fully utilize the CPU cycles of the core? Assume that the disk I/O is not the bottleneck.

Ans: 100 (computed as 1 millisecond / 10 microseconds)

(c) What is the minimum number of cores that must be running application threads in order to process all incoming load? Assume multiple threads will run on each core, as computed in the previous part, to fully saturate a core.

Ans: 10. Each request needs 10 microseconds, so each core can process 100K req/s (assuming enough threads as available), so we need 10 cores to handle 1M req/s.

(d) Assume that the average response time of each request is 2 milliseconds, which includes all user and kernel processing and wait times in various queues in the system. With an incoming load of 1M requests/second, approximately how many requests are being served by the system at any point in time?

Ans: By little's law, 2000

18. Consider a system that has a 16MB cache in the CPU (assume a simplified model of only a single level of cache), 64 byte cache lines, 4KB page size, 8GB main memory, and a TLB that can hold 1024 entries. The single CPU is running a program that accesses a large array of 4M (2^{22}) integers repeatedly. Assume the integer datatype needs 4 bytes of storage. The array is accessed sequentially from beginning to end repeatedly, and we are interested in computing the long term averaged CPU cache and TLB miss rates in steady state. Assume the CPU cache and TLB use a LRU eviction policy. Ignore CPU / cache / memory / TLB usage due to any other processes, or the kernel. Ignore prefetching or any other optimizations in hardware.

(a) What is the cache miss rate?

Ans: In the long term, the entire array of 16MB fits in cache, so the miss rate after many accesses is close to 0%

(b) What is the TLB miss rate?

Ans: 1/1024, or one miss for every 1024 array elements, which happens every time we cross a page boundary. The array spans 4K pages, so all entries will not fit in TLB. In the long term, across repeated accesses, every new page access will lead to a TLB miss, since an entry would have been evicted by the time we loop over it again.

(c) Now, suppose we turn on the huge page optimization, and all page sizes are 4MB. Recompute the CPU cache miss rate in this scenario.

Ans: Same as earlier, 0%

(d) Recompute the TLB miss rate with the huge page optimization of the previous part.

Ans: 0% now because there are only 4 pages now, so only 4 entries are needed in TLB

19. Consider a multi-threaded server that has a master-worker thread pool architecture. The server receives requests to process from clients over the network. Every request is first received by the master thread, which takes 10 microseconds to process the request, before placing it in a request buffer shared with the worker threads. A pool of 500 worker threads fetch requests from this buffer one at a time and process them. The master thread runs on one dedicated CPU, while the worker threads run concurrently and share the 3 other cores available in the system. A worker threads spends 50 microseconds executing on the CPU and 5 milliseconds waiting for disk I/O when processing each request. After completing the request processing in this manner, it goes back to fetch another request from the shared buffer (if available). Assume that the performance of the system is limited by CPU processing at the master or worker cores, and not by disk I/O or the shared buffer size or by any other resource.

(a) What is capacity of this system in req/s? That is, how many requests per second can this system handle from the clients? Also, mention the bottleneck (e.g., master CPU or worker CPUs) that is fully utilized at saturation.

Ans: 60K req/s with bottleneck at worker CPU

The master core can handle 100K req/s. Each worker core can handle 20K req/s, so 60K req/s across all 3 cores. Because the master-worker forms a pipeline, the capacity of the system is limited by the worker threads at 60K req/s. Note that we need about 100 threads per core to fully saturate the worker CPUs, and we have enough, so the number of threads is not the bottleneck.

(b) Now, suppose we optimize the worker thread code to spend only 20 microseconds on CPU and 2 milliseconds waiting for I/O. Repeat the previous question with this optimization.

Ans: 100K req/s with bottleneck at master CPU

Now each worker core can do 50K req/s, so all worker cores can together handle 150 req/s. So the master CPU now becomes the bottleneck and the capacity is now 100K req/s.

20. Consider a multi-threaded web server that receives a large number of requests from clients across the Internet. The users request web pages that are stored on disk at the server. In order to avoid going to disk repeatedly for every request, the server maintains an in-memory cache of popular files, as a linked list of web pages. The traditonal disk buffer cache used in the file system is disabled, and this web page cache is used instead. So, when the server receives a request for a web page over a socket connection, it first looks for the file in the in-memory cache, and reads it from disk only if it cannot find the web page in the linked list. A performance engineer who is given the task of optimizing the performance of the system can do one of the following things to improve system performance:

    - Optimize searching for files in the in-memory cache (say, by using a hash table instead of a linked list)
    - Increase size of in-memory cache
    - Decrease size of in-memory cache
    - Align in-memory cache structures to 64 byte boundaries

    For each of the performance problems given below, suggest which one of the above ideas is the best solution to the problem at hand. You must pick only one of the ideas for each problem, and you must briefly justify your choice.

    (a) The system has a very high CPU usage, and a profiling tool shows that searching for items in the cache linked list is taking a very long time.

    Ans: Optimize search.

    (b) The system has a very high TLB miss rate.

    Ans: Decrease the size of the cache, in order to reduce the working set size, and hence TLB usage.

    (c) The main memory is almost full, leading to lot of swapping and page faults.

    Ans: Decrease the size of the cache, in order to reduce the working set size.

(d) The main memory has enough free space, but there is a lot of disk I/O, and the disk capacity is fully utilized.

Ans: Increase the size of the cache, in order to reduce disk I/O