# COL331/COL633 Major Exam
**Operating Systems**
**Sem II, 2016-17**

Answer all 14 questions (12 pages) Max. Marks: 41

For **True/False** questions, please provide a brief justification (1-2 sentences) for your answer. No marks will be awarded for no/incorrect justification. We will penalize you if you provide extraneous information not related to the question being asked.

1. True/False: On x86 architecture, the TLB needs to be flushed on every context switch [2]

Answer: True. TLB needs to be flushed on every context switch.

VA->PA entries in the x86 TLBs are not associated with a particular address space; they implicitly refer to the current address space. Hence, every time there is a change in address space, such as a context switch, the entire TLB has to be flushed.

On a context switch, TLB is populated with VA->PA mappings of the currently scheduled process.

Modern x86 abstractions support "PCID" that let the hardware flush only selective entries.

2. The page size in our current systems is 4KB. Give an example of a workload which will greatly benefit if the page size was doubled (8KB). The workload need not be anything useful, and may just be an access pattern in the virtual address space. [3]

Answer - Example of such a workload is a sequential access pattern to an array. Doubling the page size from 4 KB to 8 KB can help in exploiting the spatial locality while accessing the elements of array. More number of sequential access to the array can be served from same page due to increased page size hence less page fault which implies better performance due to fewer TLB misses.

3. The page size in our current systems is 4KB. Give an example of a workload which will greatly benefit if the page size was halved (2KB). The workload need not be anything useful, and may just be an access pattern in the virtual address space. [3]

Answer - Example of such a workload is a strided access pattern that accesses memory in strides of 4K for example. Spatial locality is not exploited by 4KB pages in this workload, and so smaller pages imply smaller working sets, and thus less pressure on the page-cache and the L1/L2 hardware caches.

4. Give one advantage and one disadvantage of using large (4MB) pages over regular (4KB) pages. [2]

Advantages

1. More spatial locality optimizations possible hence better performance due to fewer TLB hits
2. Faster page-table walk (one level vs. two-level)

Disadvantages

1. Problem of internal fragmentation.
2. Bigger working set especially for workloads with less spatial locality, causing cache pollution. Wasted space in physical memory

5. How can an OS decide whether to use large pages or regular pages for a region of virtual memory? Give one concrete example where large pages are typically used by the OS. [3]

Answer -
Two methods.

1. OS can decide on whether to use large pages or regular pages for a region of virtual memory depending on the degree of spatial locality in the access pattern of the application. For example, Big Data applications work on terabytes of data hence large pages are suitable for such applications to exploit spatial locality. To do this, the OS needs to monitor the access pattern on the VA space, and this can be done through the hardware-set accessed bit
2. OSes provide abstractions to allow applications to explicitly request large pages.

Example : Large pages are used for mapping the kernel address space.

6. Give an example of a workload that will perform better with “Random” cache replacement policy than with the CLOCK algorithm to replace the page-cache. (A page cache is the cache for the virtual memory pages in physical memory). [3]

Answer - Consider a workload that linearly scans the virtual memory from begin to end, and does this repeatedly. This workload has almost no locality, and in fact has negative locality, i.e., a location that has been accessed most recently in the past, will be accessed very far in the future. LRU and CLOCK rely on the opposite assumption, i.e., a location that has been accessed recently in the past, is likely to be accessed shortly in the future. If the assumptions are broken, LRU/CLOCK perform worse than RANDOM.

Also correct answer : Random cache replacement policy will perform better than CLOCK algorithm to replace the page-cache in workloads where there is lesser locality (both temporal and spatial) in the access pattern. In other words, workload does not has HOT pages.

In such workloads, it is better to randomly select any page cache for replacement rather than using CLOCK algo which has its own overheads in finding a cold page for replacement.

7. In a particular system and workload (with virtual memory and swapping enabled), if four processes A, B, C, D are running together context-switching at 10ms granularity, the system runs very very slow. However if two of the processes (A and B) run for 10 seconds, and then two other processes (C and D) run for the next 10 seconds (and this is repeated every 20 seconds), the system runs *much* faster (i.e., all four jobs finish much quicker). What is the likely reason? Roughly, how much faster can the second scheme (running two processes for 10 seconds at a time) be compared to the first scheme? [3]

Answer - System runs much faster in the second case as compared to first case because in the first case, system becomes victim to thrashing (a condition in which excessive paging operations, swap in and swap out are taking place). This can happen if the sum of working sets of A, B, C and D (also called the balance set) is more than the available physical memory. However, the sum of working sets of A and B is smaller than the physical memory; and the sum of working sets of C and D is smaller than the physical memory. Thus it will be much faster to run two processes (A and B) at a time for 10 seconds, before switching to another set of processes (C and D).

The difference in performance can be significant. In the former case, the system is running at the speed of disk (if almost every access is a page fault). In the latter case, the system is running at the speed of physical memory. The access time to disk is roughly 1-10ms. The access time to memory is roughly 50-200ns. Thus the difference in performance could be anywhere between 10x-10,000x.

8. Assume you want to implement blocking locks, and you have the following abstractions available to you:

a. spinlock_t with acquire() and release() functions with standard locking semantics (but with spinning)
b. sleep() and wakeup() with the standard condition variable semantics (as in xv6).

Implement a blocking lock, i.e., implement a data-type blocking_lock_t, and two functions blocking_acquire() and blocking_release() using the spinlock and sleep/wakeup abstractions. [4]

Answer:

```
 struct blocking_lock_t {
          int  locked;
          struct spinlock_t spinlock;
}

void init(struct blocking_lock_t *l) {
          l.locked = 0;
}

void blocking_acquire(struct blocking_lock_t *l) {
          spinlock_acquire(&l->spinlock);
          while (l->locked != 0) {
                    sleep(l, &l->spinlock);
          }
          l->locked = 1;
          spinlock_release(&l->spinlock);
 }

 void blocking_release(struct blocking_lock_t *l) {
          spinlock_acquire(&l->spinlock);
          l->locked = 0;
          wakeup(l);
          spinlock_release(&l->slock);
 }
```

9. Consider the following function ‘remove_node’ to remove a key from a singly-linked list. It assumes that the key is not present in the head (first node) of the linked list. It returns the removed node if found. It returns NULL if the key is not found.

```
struct node {
   int key;
   struct node *next;
};

node *
remove_node(node *head, int key)   //assume that the head node does not contain ‘key’
{
   while (head) {
      if (head->next != NULL && head->next->key == key) {
         node *ret = head->next;
         head->next = head->next->next;
         return ret;
      }
   }
   return NULL;
}
```

a. If multiple threads execute this code in parallel, what are some bad things that can happen? Assume that it has been ensured that the concurrent threads will try and search/remove the same key. Also assume that the keys in the list are unique. Give one example of an undesirable situation. [2]

Answer: Some bad things that can happen-

A thread can remove a wrong node. Suppose two threads having same key as input, execute the if condition in parallel, both them of finds head->next to be not null and head->next->key to be same as key. Now thread 1 gets to run the if case body, increment head-> next and removes the node having matching key. Correct behaviour till now. Now when thread 2 gets to run, it will also increment the next pointer but it will remove the wrong node whose key does not match with the input key.

There can also be race condition in the above code when multiple threads execute in parallel.

b. How can you fix the problem using coarse-grained locks, i.e., one lock for the entire list? Write the code that you will need to change/add. [1]

```
struct lock l;

struct node {
    int key;
    struct node *next;
    struct lock *l;
};

node *remove_node(node *head, int key)   //assume that the head node does not contain 'key'
{
    acquire(&l);
    while (head) {
        if (head->next != NULL && head->next->key == key) {
            node *ret = head->next;
            head->next = head->next->next;
            release(&l);
            return ret;
        }
    }
   release(&l);
    return NULL;
}
```

c. How can you fix the problem using the compare-and-swap instruction cmpxchg(), as discussed in class? Write the code that you will need to change/add. [3]

Answer-

```
node *
remove_node(node *head, int key)   //assume that the head node does not contain 'key'
{
        while(head) {
    retry:
                node *next = head->next;
                If (next !=NULL && next->key == key) {
                    node *ret = next;
                    node *new_next = next->next;
                    if (cmpxchg(&head->next, ret, new_next) == ret) {
                        return ret;
                    } else {
                        goto retry;
                    }
                }
                head = head->next;
        }
        return NULL:
}
```

10. Sleep/wakeup

a. Almost all calls to the sleep() function in xv6 are inside a loop, e.g.,

```
while (...) {
    sleep(..., ....);



}
```

Why? [1]

Answer: It is because when process/thread is notified to wakeup by some other process/thread, the condition needs to be checked again to ensure that any other thread which also got woken up did not falsify the condition. In other words, the operation of waking up and reacquiring the lock are not atomic, and other processes may get to run  in between; so the condition needs to be checked again upon returning from sleep().

b.  Look at line 2683 in xv6 code

```
2683	:	if(!havekids || proc−>killed){
```

If this condition is true, the function returns, else it goes to sleep.  What is the significance of checking (!havekids)?  What is the significance of checking (proc->killed)?  If the check on (proc->killed) was removed, would the code remain correct?  What are some bad things that may happen?  [3]

Answer: The significance of checking (!havekids) is that if the process doesnot have any child process, i.e. process is not waiting on some child process then process must not be made to sleep.

Checking proc->killed: if the process has been killed, it should not go back to sleep, but should return, so that the caller can call exit on this process.

Some bad things that may happen if the check proc->killed is removed are-  the process may sleep indefinitely, till one of its children exit.  Thus the time gap between the process being killed by the user, and the process actually exiting and freeing up its resources can be unbounded. Checking killed ensures bounded time interval between killing a process and its exiting.

11. When not using logging, and using ordering of writes to disk, to ensure crash recovery, what are the types of inconsistencies that may still happen?  Give some examples of what inconsistencies can still happen, and what inconsistencies are prevented?  [2]

Answer: When not using logging and using ordering of writes to disk, to ensure crash recovery, the type of inconsistencies that may still happen are “space leaks”.

For example, while creating a file, the ordering of operations is such that firstly the data is written to the disk block and then the size in the inode of directory is updated. If the crash happens in between these two operations then there is a problem of memory leak as the size would not have been yet updated.

Another example is-
$ mv a/ foo b/ foo (Ordering of writes to disk is such that firstly a file is created in directory b and then the file foo is removed from a to avoid data loss.

Inconsistencies which are prevented are data loss and inconsistencies in the FS tree structure, which may also lead to security issues.

12. True/False. When not using logging (and using ordering as in the question above), the order in which locks are acquired on blocks and the order in which they must be flushed to disk, is always identical. If True, explain why. If false, give counter-example. [3]

Answer: The order in which locks are acquired on blocks/buffers and the order in which they must be flushed to disk is not usually identical. For example, in xv6, the order in which locks are acquired is "top down": first the parent buffer's lock is acquired (as in the FS tree) before the child buffer's lock is required. On the other hand, the flushing order to ensure crash recovery should be opposite : the child buffer should be flushed before updating the parent poiner.

13. In logging-based crash recovery, explain the disadvantages if the transaction size is too small (e.g., one transaction per filesystem operation)? Also, explain the disadvantages if the transaction size is too big (e.g., one transaction very two hours)? [2]

Answer: In logging based crash recovery, if the transaction size is too small, then there will be lot of commit operations. This will increase the amount of disk seek time and rotational latency, one seek plus rotational latency per commit operation. Hence, system performance wills reduce as disk being a magnetic device is slow.

On the other hand, if the transaction size is too big say one transaction per two hours , then in case of a crash user will be under the impression that the changes to disk which he/she made long back must have been reflected but they actually are not.

14. After having done the OS course, what was your favourite OS topic? [1]

a. OS abstractions (e.g., system call interface design)
b. Instruction Set Architecture for implementing OS abstractions
c. Traps/Interrupts and privilege separation by architecture
d. Virtual memory (segmentation, paging and swapping)
e. Scheduling
f. Concurrency control (locking, condition variables, etc.)
g. Device drivers
h. Filesystem design
i. Crash recovery
j. None of the above. In this case, mention your favourite topic (if any).