CSL373/CSL633 Major Exam
Operating Systems
Sem II, 2013-14

Answer all 11 questions (14 pages) Max. Marks: 62

For **True/False** questions, please provide a brief justification for your answer. No marks will be awarded for no/incorrect justification.

1. Does a logging-based filesystem (like ext3) make sense if the buffer cache is write-through. State your assumptions about how logging will be implemented with a write-through buffer cache to answer this question. [3]

Yes, it makes sense. logging provides atomicity across multiple disk writes. Atomicity guarantees may be required both for write-through and write-back filesystems.

2. In xv6, consider the following situation:
   a. A thread (say thread1) is executing on CPU0, and acquires spinlock X
   b. A timer interrupt occurs and the thread1 gets context-switched out (while holding X)
   c. Another thread (say thread2) starts executing on CPU0 and tries to acquire spinlock X, and spins (because thread1 is holding X)

Assume that threads in xv6 can share address space both within the kernel and in user-space (i.e., multiple processes can map the same pages in the user address space and access them concurrently). Answer the following questions:

(i) Based on this sequence of events and your knowledge of the xv6 kernel, can you say if the thread was executing in user-space or kernel-space at the time of the context-switch? [3]

User-space. In kernel space, a spinlock acquisition disables interrupts.

(ii) How long will thread2 spin on CPU0? Why? [2]

Either till thread1 will get scheduled on CPU1 and releases the lock or till thread2's quanta of time got over and gets context switched out by the timer interrupt.

(iii) If there are 100 threads that all acquire spinlock X, and the xv6 scheduler is round-robin, then what is the problem? [2]

If a thread gets preempted while holding the lock, any other thread trying to acquire lock will just spin and waste resource.

Worst case scenario: All the thread acquires the lock and every thread, once it acquired the lock, gets preempted while holding the lock
Once the thread holding lock got preempted, 99 other threads will do busy waiting for their scheduling quanta and then only the thread holding the lock will get chance to run.

(iv) Give two potential solutions to this problem. Your solution could involve change in the xv6 design/abstractions. Discuss which solution is better in what situations. [4]

Note: 1. Blocking locks will solve the problem of resource wastage but it may perform poor when the critical sections are small.
2. Spin and block. Spin for x amount of time and then if not able to acquire then yield/sleep also doesn't solve the problem. It only reduces the wastage of time by reducing the amount of time the thread spins.

1. Spin and yield to thread holding the lock. Atomically set the threadId before acquiring the lock(using CAS). Spin for x amount of time, and if not able to acquire the lock, yield the CPU to the threadId which is set atomically. xv6 needs to have yieldTo(threadId) API.

2. Change in scheduler: Scheduler may notify the thread before switching it out. The thread will get chance to release the locks and then yield. (exokernel style)

3. Consider the following pseudo-code for exit/wait synchronization (only relevant parts shown, this pseudo-code is incomplete) using sleep/wakeup primitives, taken from xv6 lines 2354-2430:

```
void exit(void) {
    // release some resources, e.g., fds
    // wakeup(my parent process)
    // set my state to ZOMBIE
}

int wait(void) {
   // iterate over all my child processes
   // if find a child process that is ZOMBIE, free its resources and return its pid
   // if we find a child process that is currently not ZOMBIE, sleep(myself)
}
```

Answer the following questions:

a. What happens if the child calls exit() before the parent calls wait()? [3]

exit:
wakeup would not do anything
will set the state of the child to ZOMBIE.

wait:
whenever called in future will find a ZOMBIE child and free it up.

[Those who say that wakeup will be lost and parent would keep sleeping will get zero marks. The wakeup will be lost but there is no problem due to this.]

b. What happens if the parent calls wait() just after the child calls wakeup()? [3]

It will work fine. As the ptable.lock is already acquired by child so parent would wait for child to release the lock (which happens only after setting the state to zombie.)

[Those who say that wakeup will be lost and parent would keep sleeping will get zero marks]

4. When an executable program is loaded into memory using the exec() system call, it is possible that only a few pages are loaded initially and the rest are demand-paged.

a. What are the advantages of demand paging? [2]

1. Process creation takes less time (Only few pages have to be loaded)
2. Uses minimal memory (Since all the pages of a program that are accessed will be demand paged)
3. Potentially many processes can run (Since each process now occupies minimal memory)
    a. Similar point : May increase speed of programs if their pages are always kept in the main memory instead of disk (Thrashing)

[Any 1 point 1 mark, At least 2 points 2 marks]

b. Is it possible that the overall performance of the system could have been better if demand paging was disabled, i.e., all pages of the executable are loaded apriori at load time? Give an example to justify. [4]

1. Demand paging is not practical in embedded systems where area is a serious constraint.
2. In some embedded systems, the programs are known apriori and the code portions they execute are also known apriori. In such cases, it is better to avoid extra MMU cost and make the paging hard coded.
3. If a process accesses a lot of pages and memory is not a constraint, it is better to allocate all the pages in the beginning. (Since handling page fault is a costly operation)

[Point number 2/3 with proper explanation - 4 marks]
[Point number 2/3 with half baked explanation -2 marks]
[Point number 1 - 2 marks]

c. Explain why the copy-on-write optimization allows UNIX to separate process creation into two system calls, namely fork and exec? [3]

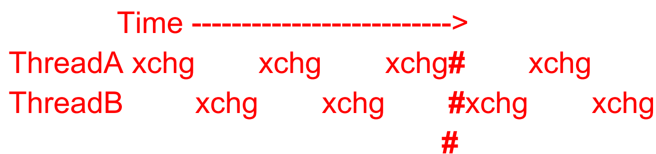
Due to copy on write optimization, the code and data section of the parent and the child process remain shared. Whenever any process tries to write to the shared section, a new page is allocated and the corresponding process proceeds.

So, due to COW optimization, the fork system call is very fast. Now if the child has to exec a different file, it can do it without interfering with the execution of the parent process. This allows the fork+exec combination almost as fast as a combined process-execute command on Windows.

The task of fork system call is now to inherit the file descriptors and other properties.
The task of exec system call is to load a new code and data section.

[Something on copy on write optimization - 0.5 marks]
[Explained the fork/exec system call - 2 marks]
[Mentioned function-reuse or fast-usage - 3 marks]

5. True/False: - The spinlocks implemented using the xchg() instruction (as in xv6) obey arrival order, i.e., if thread A reaches the acquire function before thread B (by reaching the acquire function, we mean the first execution of the atomic xchg instruction by the thread), then thread A will definitely obtain the lock before thread B (for example, even if the lock was held by another thread C). Explain. [3]
False. Which thread gets the lock is decided by the race condition of when does the thread C release the lock.



Thread C releases lock.Thread B will now acquire lock.

Note : Even otherwise, the architectural property of the system also plays a role in deciding which core gets the cache line belonging to the lock variable.

[No part marking. Correct answer - 3 marks]

6. Fix the following program code such that all concurrent calls to the function foo() by multiple threads, are effectively serialized. [2]

```
void foo(void) {
   struct lock mutex;
   acquire(&mutex);
   // access a global variable
   release(&mutex);
}
```

```
struct lock mutex = null;

void foo(void) {
   if (mutex==null) {
      init_lock(&mutex);
   }
   acquire(&mutex);
   // access a global variable
   release(&mutex);
}
```

[Recognised that mutex is not initialised - 1 mark]
[Any correct code that makes lock a global variable - 2 marks]

7. The hardware provides the “accessed bit” per page to implement a cache replacement algorithm that accounts for access patterns, e.g., CLOCK.

a. CLOCK is also called a 1-bit approximation to LRU. Why? [2]

The accessed bit acts as a one-bit timestamp, which divides pages into two sets: accessed recently, and not accessed recently.

b. What is the advantage of using a two-hand clock over a single-hand clock? Explain clearly with example. [2]

Two-hand clock is useful for systems with large memories. Less CPU consumption for replacement, tunable interval for judging if a page was accessed recently or not.

8. Security: Consider a UNIX system, in which a special ‘root’ user has all privileges. Also assume that all common utility files (e.g., executables, configuration files, etc.), are owned by the ‘root’ user. Answer the following questions:

The executable file ‘/bin/ls’ is one such file owned by the root user. As you know this executable displays the contents of a directory.

a. Should the setuid bit on the ‘/bin/ls’ executable be one or zero? What are some bad things that can happen if the setuid bit of the ‘/bin/ls’ executable is set to one? Does it matter? [3]

setuid bit should NOT be set on the /bin/ls executable. If /bin/ls has setuid bit set, then a user can invoke /bin/ls on root’s directory to see files that it was not supposed to see.

Wrong answer: if the answer involves modifying ls

b. A special program called ‘sudo’ allows a non-root user to execute commands using root privileges. The commands to be executed are provided as arguments to the ‘sudo’ program. Can the sudo program be implemented without using the setuid bit? Who is the owner of the ‘sudo’ executable? [3]

Nope. If the sudo program does not use the setuid bit, the user CANNOT perform operations that only the root is supposed to do. The owner of the ‘sudo’ executable is root.

9. Many operating systems provide a ‘disk defragmenter’ utility. This utility can be invoked by the user to reorganize her disk contents, for better future disk performance.

a. What kind of reorganization could this tool do, to try and improve future disk performance? Assume that the filesystem remains the same. [3]

Locality optimizations.
Optimize for Sequential of the same file: Move the blocks so that they’ll be contiguous.
Optimize for related files: Move files which are accessed together closer to each other.
Optimize for frequentially modified files and less frequently modified files and append only files.

b. Assuming best possible implementation, what is the approximate time it will take to run such a tool on a disk with a filesystem with 100GB worth of files, in the worst case? Assume that you have 500GB RAM in your system. Show how you arrived at your answer. Does your answer depend on the filesystem being used? [4]

Generic:
Read all 100GB of files into RAM:random reads bandwidth+random seek latency(slow)
Time: = 100GB * random_read_bandwidth
Group the related files together
Time: negligible (depends on no of files0
Remove all the files
Time: latency + metasize*bandwidth
Write the files from RAM into the filesystem: full sequential write bandwidth
Time: 100GB*write_bandwidth

Yes: random read bandwidth depends No of seeks which depends on the filesystem being used. (if file system internally guarantee that the files are sequential or not)

Solution:
Read all the blocks from the filesystem sequentially: full read sequential bandwidth
Time: 100GB*sequential_read_bandwidth
Sort the blocks by file
Time: k * n * log n.
Write lethe blocks from RAM to the filesystem: full write sequential bandwidth:
Time: 100GB*sequential_write_bandwidth
No: it doesn’t depend on filesystem being used

Assuming sequential write bandwidth of 100MBps, this will take around a 1000 seconds (or around 15-20 minutes)

10. Consider the following sequence of shell commands on the Linux/ext3 filesystem:

```
$ echo hello > a &
$ echo world > b &
$ ls > c &
```

Notice that the ‘&’ symbol means that all these three commands execute concurrently, and can finish in any order. Which of the following are consistent states after a crash, assuming ext3 journaled mode? In other words, which of these filesystem states are possible after a crash on Linux ext3 journaled mode? Explain briefly. [6]

1. File ‘b’ exists, file ‘a’ exists, and file ‘c’ is empty. [2]

ls could have executed before the echo commands, so this is a consistent state.

2. File ‘b’ exists, file ‘a’ does not exist, and file ‘c’ contains two entries (for ‘a’ and ‘b’). [4]

This is not possible (inconsistent). Clearly, ls executed after the two echo commands. Either ‘ls’ would be a part of the same transaction as the echo commands, or it will be a part of the next transaction. It cannot be a part of a previous transaction (as it was logically executed after the echo commands). This situation cannot happen if ‘ls’ belonged to the same transaction. It can also not happen if ‘ls’ belonged to the next transaction.

11. Give examples (qualitatively) of situations when :

a. Big reader locks perform better than reader-writer locks [2.5]

When the access type to the shared data is mostly read and rarely there is an update. And concurrency is high.

b. Reader-writer locks perform better than big reader locks [2.5]

Accept: if application has large memory footprint, and large reader locks result in large cache pollution.

Also accept: When there are multiple readers and there is no restriction that writes would be rare then big reader locks perform poorly.