**CSL373/CSL633 Major Exam Solutions**
**Operating Systems**
**Sem II, 2012-13**
**May 6, 2013**

Answer all 8 questions Max. Marks: 56

**1. True or False.** Give reasons and/or brief explanation. No marks for incomplete/wrong explanation.

a. On xv6, the walkpgdir() function gets executed on every memory access by the user process. [1 mark]

False. For every memory access, the hardware walks the page table (or consults the TLB cache). walkpgdir() is only used by the kernel to simulate the same behaviour while adding/removing pages to a user process address space.

b. For most desktop applications, using huge pages (e.g., 2GB pages) will result in better overall system performance. [1 mark]

False. Huge pages are only useful if applications have large contiguous memory footprints. This is not true for most desktop applications.

c. An in-memory filesystem, RAMFS, as discussed in the xsyncfs paper, will always have better performance than any on-disk filesystem for a disk-intensive benchmark. [1 mark]

True. Each RAMFS write operation just writes to memory. On-disk filesystems write to disk. Disk I/O is orders of magnitude slower than memory access.

d. In a ext3 journaling file system, a transaction can be closed at any time. [2.5 marks]

True. A transaction can be closed at any time. Subsequent write operations will belong to a new transaction. Operations that had already started will belong to the previous transaction. Thus, the previous transaction will have to wait for all these operations to complete before installing it to disk.

Hence, a transaction can be closed at any time, but it must wait for ongoing write operations to complete before it can install it to disk.

e. In xv6, a file create can fail even if the disk is not full. [2 marks]

True. This can happen if:

i. The inode number limit is reached.
ii. A filename with the same name already exists, or any other such reason.

Full marks if any of these is mentioned.

f. On a FAT32 filesystem, corruption of one disk block could potentially lead to the loss of complete file data. [2 marks]

False. By storing redundant copies of the File Allocation Table (FAT), we guard against loosing data due to single block corruption. If redundant copies are not stored, file data could be lost because FAT uses linked files.

g. The best possible filesystem layout for a read-only filesystem (which is written only once in the beginning) is contiguous allocation. Explain why or why not. [2.5 marks]

True. Because this is write-once filesystem, there will be no dynamic file creations and no file growth. A contiguous filesystem provides the best possible sequential and random I/O performance. The demerits of contiguous file system do not exist, as it is write-once.

h. When using a write-through buffer cache, one does not need to worry about the order of disk writes (from a filesystem consistency perspective across power reboots). Explain why or why not. [2 marks]

False. The order of disk writes is important to ensure that filesystem invariants are maintained across power reboots. For example, an invariant could be that there should be no dangling pointers. Further meta-data writes to the disk device should include write barriers to avoid unexpected behaviour due to disk controller caching.

i. The “First Fit” algorithm is used when allocating pages to processes. [1 mark]

False. The first fit algorithm is used for allocating variable-sized memory (e.g., for malloc). It is not used for fixed-size pages.

**2**. In xv6, the kernel stack (kstack) of a process is limited by KSTACKSIZE (4096 bytes). List at least three invariants that the kernel maintains to ensure that kstack never overflows. i.e., if one of those invariants is violated, kstack could potentially overflow. [5 marks]

a. There is a limit on the number of nested external interrupts. On xv6, this limit is 1, i.e., the interrupt flag is cleared while executing an external interrupt handler.

b. There is a limit on the number of nested external exceptions. for example, a page fault exception should not occur while executing a page fault handler, else we could potentially have infinite nesting.

c. Stack variables and arrays are avoided. Instead, memory is heap allocated, wherever required in the kernel.

Deduct 1.5 marks if any of these answers not given.

**3**. The L-1 cache of a processor could either be physically indexed (i.e., indexed using physical addresses), or virtually indexed (i.e., indexed using virtual addresses). List the primary advantage of using a virtually addressed L1 cache. List the primary advantage of a physically addressed L1 cache. [3 marks]

The primary advantage of a virtually addressed L1 cache is that there is no need to go through paging hardware while accessing L1. This improves L1 access latency.

The primary advantage of physically addressed L1 cache is that there is no need to flush the L1 cache across process context switches.

**4**. Professor X really liked the xsyncfs paper, but found it a bit “incomplete”. He suggests that external synchrony should be implemented over a cluster of machines, to achieve its real potential. He is developing a new filesystem called “networked externally synchronous file system”, or *nxsyncfs*. In nxsyncfs, an output is not considered *external*, until it leaves to the external network. In other words, all communication between two hosts within the cluster is considered *internal*. Assuming that a cluster of machines can only be observed through the external network interface (assume there is no console access to these machines), answer the following questions:

i. What will be your filesystem sync policy on nxsyncfs? i.e., when will you insert fsync calls? Does nxsyncfs give more optimization opportunities over xsyncfs? Briefly explain. [2 marks]

Now, disk buffers need to be flushed to disk only on “external output”. Hence, if there is an output on an external network interface by any machine, it will check if there is a causal dependency between this external output and a preceding write to disk. If so, it will call fsync on the host on which this preceding write was executed.

Yes, this provides further optimization opportunities, as now larger group commits are possible. A commit only needs to be performed on a write to the external network. In contrast, if xsyncfs was used, a commit would have to be performed on every communication between two hosts in the cluster (assuming dirty buffers existed).

ii. What kind of data structures and mechanisms will you need to implement for nxsyncfs? Assume that all xsyncfs mechanisms are already in place. Only emphasize the extra structures and mechanisms that you will need to add over xsyncfs, to take advantage of nxsyncfs semantics? [3 marks]

At each host:
1. Maintain information on whether my buffers are dirty or not. Also maintain information on whether a host which had dirty buffers sent a causally following communication to me or not. Maintain a list of such hosts as “dirty hosts”.
2. On a network output, check to see if the destination host is internal or external. If it is external, call fsync() locally (if my buffers are dirty) and instruct all dirty hosts to also call fsync() [this may involve a request/response network mechanism]. A network packet should be sent to external network only after fsync() is called on all such hosts.
3. If the network output is internal and the network output follows a causally preceding disk write, then add this host to the list of dirty hosts in the destination.

iii. Give an example of an application that will show significant improvements with nxsyncfs over xsyncfs. i.e., the performance of the application should significantly improve if nxsyncfs is used in place of xsyncfs.  [3 marks]

Any benchmark which involves a large amount of disk activity across multiple hosts, and a large amount of internal network activity and relatively smaller amount of external network activity. Some examples:

1. Web application server backed by database and file servers. There is a lot of communication between the application server, database server, and file server, but a group commit will happen only when an external packe to the client is sent.

2. Data analytics or scientific applications processing a large amount of disk data.

...

**5. Producer-consumer with semaphores**

Consider the following functions, produce(item), and consume() for a shared queue, shared between multiple producers and multiple consumers. As discussed in class, semaphores are used to synchronize access to this shared queue.

```
void produce (item):
        sema_down(mutex);
        sema_down(holes);
        insert_item(item, buffer);
        sema_up(items);
        sema_up(mutex);


item consume(void):
        sema_down(mutex);
        sema_down(items);
        item = remove_item(buffer);
        sema_up(holes);
        sema_up(mutex);
```

In this code, the semaphore “mutex” is used for mutual exclusion, semaphore “holes” is used to count the number of empty slots in the buffer, and semaphore “items” is used to count the number of items in the buffer.

a. What should be the initial values of the semaphores mutex, holes, and items? Assume that the buffer size is N. [1.5 marks]

mutex: 1
holes: N
items: 0

Assuming the buffer is initially empty, it has N holes and 0 items. mutex is initialized to 1, so it acts as a lock.

b. Look at this code closely and answer if this code is correct. If you think it is correct, briefly explain the invariants that prove it correct. If you think it is incorrect, explain why and fix it so that it becomes correct (with brief explanation as before). [4.5 marks]

This code is incorrect, it has a deadlock.

Consider the following sequence (assume 1 producer, 1 consumer):
Intially, holes = N, items = 0
1. consumer does sema_down(mutex)
2. consumer does sema_down(items) and blocks.
3. producer does sema_down(mutex) and blocks.
Deadlock!

Correct code:

```
void produce (item):
        sema_down(holes);
        sema_down(mutex);
        insert_item(item, buffer);
        sema_up(mutex);
        sema_up(items);


item consume(void):
        sema_down(items);
        sema_down(mutex);
        item = remove_item(buffer);
        sema_up(mutex);
        sema_up(holes);
```

**5'.** Consider the ext3 journaling filesystem. Answer the following questions:
a. How does ext3 ensure a low runtime overhead, even though a disk write results in at least two separate disk writes (one to log, and one to FS tree). [1.5 marks]

1. Writes to log are coalesced by using compound transactions. Writes become sequential.
2. Application of log to FS tree is done asynchronously

b. Consider a situation where filesystem compound transaction X has started committing. What happens to the transactions (or atomic operations) that are already ongoing? Do the updates of those operations get reflected in the next compound transaction (say transaction Y), or do they get reflected in the previous transaction (transaction X)? [2 marks]

The updates of these operations are reflected in transaction X. Committing can start, i.e., updates can be written to log but a final commit record (or an update of head/tail of log) will only happen after all ongoing operations (which started before X closed) have completed.

c. What happens to the operations that were started after transaction X started committing but before the commit record was written? Do the updates by these operations get reflected in the previous transaction (transaction X) or the next transaction (transaction Y)? [2 marks]

The updates of these operations are reflected in transaction Y. If a commit has started, then the previous transaction has been closed. To limit the size of transactions, we do not allow subsequent operations to go to the previous transaction. These operations are made a part of transaction Y.

d. While transaction X is committing (i.e., commit record has not yet been written), what happens if some operation in transaction Y writes to a disk block which is also a part of transaction X? How does ext3 ensure correct behaviour? [3.5 marks]

A copy of that block should be made by transaction Y in memory, and operations by transaction Y should be done on this new copy. The old copy (in memory) can be used for committing transaction X. Once transaction X has finished comitting (commit record is written), the old copy should be discarded and the new copy (made by transaction Y) should be made the master copy.

**6.** Implement reader-writer locks using spinlocks and sleep/wakeup. You must not use any other synchronization mechanism. You should also not assume specific architecture instructions like xchg, etc. Your implementation should be efficient.

No scheduling requirements: The scheduling of read/write locks can be arbitrary, i.e., if a reader (potentially many) and a writer are both waiting for a lock, any one can win. Also, if a lock is being held in read mode, other readers may be allowed to acquire it, even if a writer is waiting.

You have to implement: struct rwlock, read_acquire(), read_release(), write_acquire(), write_release() functions.

You are allowed to use: spinlock_acquire(), spinlock_release(), sleep(), wakeup(). [4 marks]

```
void read_acquire(struct rwlock *l) {
  spinlock_acquire(&l->slock);
  while (l->state == WRITE_LOCKED) {
    sleep(l, &l->slock);
  }
  l->locked = READ_LOCKED;
  l->reader_count++;
  wakeup(l);
  spinlock_release(&l->slock);
}

void read_release(struct rwlock *l) {
  spinlock_acquire(&l->slock);
  l->reader_count--;
  if (l->reader_count == 0) {
    l->state = UNLOCKED;
  }
  wakeup(l);
  spinlock_release(&l->slock);
}
```

```
void write_acquire(struct rwlock *l) {
  spinlock_acquire(&l->slock);
  while (l->state != UNLOCKED) {
    sleep(l, &l->slock);
  }
  l->locked = WRITE_LOCKED;
  spinlock_release(&l->slock);
}
void write_release(struct rwlock *l) {
  spinlock_acquire(&l->slock);
  l->state = UNLOCKED;
  wakeup(l);
  spinlock_release(&l->slock);
}
```

**7.** Security: Explain how submit-pintos script is able to both write to my (sbansal's) home as well as to your (the person who is submitting) home. [3 marks]

Using the setuid bit in the access control bits of the submit-pintos script.

The setuid script sets the euid to sbansal, and uid to the invoker.

The script is capable of switching between its euid and uid through the seteuid() system call (or equivalent) to achieve this functionality.

**8.** How does an OS detect Thrashing? [1.5 marks]. How does it deal with it? [1.5 marks]

Thrashing can be detected by measuring the page fault frequency. A high page fault frequency (due to swapping) implies thrashing. It can be avoided through better scheduling. For example, a set of processes whose working set is likely to fit in the available RAM is run together for an extended time (say 1-5 seconds), and then swapped out together to allow another group of processes to get swapped in. This reduces the rate of page faults, and ensures faster overall progress.