Final Examination
CS 111, Winter 2020
3/18/2020, 11:30AM – 2:30PM

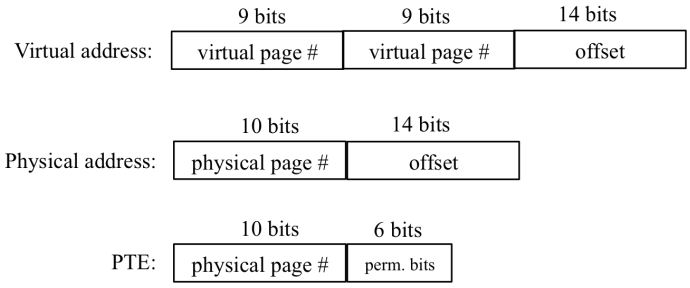
Name: _____ Student ID: _____

One single-sided cheat sheet is allowed.

I understand this is a closed book, closed notes test and I herby attest that I have completed this exam solely by myself.

Signature: _____

1. Paging. Consider a memory architecture using two-level paging for address translation. The format of the virtual address, physical address, and PTE (page table entry) are bellow: **15points, 3 points each question**



(a) What is the size of a page?
A: 2^{14} bytes = 16384 bytes = 16 KB

(b) What is the size of the maximum physical memory?
A: 2^{24} bytes = 16 MB

(c) What is the total memory needed for storing all page tables of a process that uses the entire physical memory?

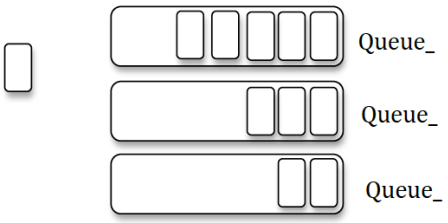
*A: There are 2^{10} = 1024 physical pages. There is one page table at the first level, and up to 2^{9} = 512 page tables at the second level. Since the physical address is 3 bytes, the size of the first level page is 2^{9} * 3 bytes = 1,536 bytes. Furthermore, the PTE is 2bytes, so the size of a second level table is 2^{9} * 2 bytes = 1024 bytes. All in all, the page tables use 1,536 bytes + 512 * 1024 bytes = 528,384 bytes of memory. (Notes: We gave full credit to answers assuming that the entries at the first level page are 2 bytes, as well. Indeed, the last 9 bits of an address to a page table are typically 0, and don't need to be stored.)*

(d) Assume a process that is using 512KB of physical memory. What is the minimum number of page tables used by this process? What is the maximum number of page tables this process might use?
A: The process uses 512KB / 16 KB = 32 physical pages. Since a second level page can hold up to 512 PTEs, in the best case scenario we use only 2 page tables: 1st level page + a 2nd level page. In the worst case, the process may use a little bit of every physical page (e.g., 0.5 KB of each physical page), and all page tables will be populated. Thus, the process ends up using 1 + 512 = 513 page tables. (Note: We have also given full credit to people who assumed that the process fully uses each physical page. In this case the answer is 1 + 32 = 33 page tables.)

(e) Assume that instead of a two-level paging we use an inverted table for address translation. How many entries are in the inverted table of a process using 512KB of physical memory?

A:The inverted table maintains one entry per physical page. In the worst case, the process uses all physical pages, which yields 1024 entries. In the best case, the process fully uses each physical page, which yields 32 entries. (Note: We gave full credit to people who only answered: 32 entries.)

2. Synchronization. **15 points**



Consider a set of queues as shown in the above figure, and the following code that moves an item from a queue (denoted "source") to another queue (denoted "destination"). Each queue can be both a source and a destination.

```
void AtomicMoveItem (Queue *source, Queue *destination) {
   Item thing; /* thing being transferred */
   if (source == destination) {
      return; // same queue; nothing to move
   }
   source->lock.Acquire();
   destination->lock.Acquire();
   thing = source->Dequeue();
   if (thing != NULL) {
      destination->Enqueue(thing);
   }
   destination->lock.Release();
   source->lock.Release();
}
```

Assume there are multiple threads that call AtomicMoveItem() concurrently. (5 points each question)

(a) (3 points) Give an example involving no more than three queues illustrating a scenario in which AtomicMoveItem() does not work correctly.

A: If one thread transfers from A to B, and another transfers from B to C and another from C to A, then you can get deadlock if they all acquire the lock on the first buffer before any of them acquire the second.

(b) (6 points) Modify AtomicMoveItem() to work correctly.

A: One solution to solve the problem is to impose a total order on how locks are acquired/released. The following code uses the source/destination object addresses to impose such an order, i.e., the source/destination object with a lower address acquire the lock first (the modified code is in bold):

```
void AtomicMoveItem (Queue *source, Queue *destination) {
        Item thing; /* thing being transferred */
        if (source == destination) {
                return; // same queue; nothing to move
        }
        if (source > destination) {
                source->lock.Acquire();
                destination->lock.Acquire();
        } else { // destination < source
                destination->lock.Acquire();
                source->lock.Acquire();
        }
        thing = source->Dequeue();
        if (thing != NULL) {
                destination->Enqueue(thing);
        }
        if (source > destination) {
                source->lock.Release();
                destination->lock.Release();
        } else { // destination < source
                destination->lock.Release();
                source->lock.Release();
        }
}
```

(c) (6 points) Assume now that a queue can be either a source or a destination, but not both. Is AtomicMoveItem() working correctly in this case? Use no more than two sentences to explain why, or why not. If not, give a simple example illustrating a scenario in which AtomicMoveItem() (given at point (a)) does not work correctly.

A: The code presented at point (a) will work correctly in this case, as it cannot lead to deadlock. This is because AtomicMoveItem() will always acquire the lock of the source, first and the lock of the destination second, (Next, we give a "proof"; this proof wasn't required for receiving full score.) The fact that AtomicMoveItem() always acquires the source lock first guarantees that you cannot end up with a cycle. Indeed, assume this is not the case, i.e., thread T1 holds the lock of queue1 and requests the lock of queue2, T2 holds the lock for queue2, and requests the lock of queue3, ..., Tn holds the lock for queuen and waits for the lock of queue1. Since T1 holds the lock of queue1 but not queue2, it follows that queue1 is a source queue, while queue2 is a destination queue.

Furthermore, since Tn holds the lock of queuen but not queue1, it follows that queue1 is a destination queue. But queue cannot be at the same time source and destination, which invalidates the hypothesis that the pseudocode can lead to deadlock.

3. One of the innovations of the BSD file system is that it tries to allocate large files in long contiguous chunks. Assuming that a disk transfers at a peak rate of 200 MByte/s, and that a combined seek and rotation take, on average, a total of 20 milliseconds. What is the minimum size of each contiguous run of a large file, in order to achieve 80% of peak transfer rate for large files when they are accessed sequentially? **5 points**

   A: The seek/rotation takes 20 milliseconds, so to achieve a 80% peak transfer rate, we need to spend 80/20 of the seek/rotation time transferring data, which is 80 ms: 80 x 10^{-3} seconds x 200 x 10^{6} bytes/seconds = 16,000 x 10^{-3} bytes = 16 Mbytes

   This would also give us 12MB

4. Large files. We have seen different filesystems that support fairly large files. Now let's see just how large a file various types of filesystems can support. Assume, for all of the questions in this part, that filesystem blocks are 4 KBytes. **20points**

   i) (4 points) Consider a really simple filesystem, *directfs*, where each inode only has 10 direct pointers, each of which can point to a single file block. Direct pointers are 32 bits in size (4 bytes). What is the maximum file size for *directfs*?

   A: The maximum directfs file size is 10 x 4 KByte = 40 KByte

   ii) (4 points) Consider a filesystem, called *extentfs*, with a construct called an extent. Extents have a pointer (base address) and a length (in blocks). Assume the length field is 8 bits (1 byte). Assuming that an inode has exactly one extent. What is the maximum file size for *extentfs*?

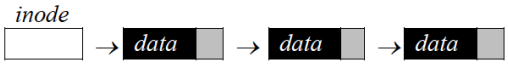
   A: The maximum extentfs file size is (28 – 1) x 4 KByte = 255 x 4 KByte = 1 MByte

   iii) (4 points) Consider a filesystem that uses direct pointers, but also adds indirect pointers and double-indirect pointers. We call this filesystem, *indirectfs*. Specifically, an inode within *indirectfs* has 1 direct pointer, l indirect pointer, and 1 doubly-indirect pointer field. Pointers, as before, are 4 bytes (32 bits) in size. What is the maximum file size for *indirectfs*?

*A: The maximum indirectfs file size is (1 + 1024 + (1024 x 1024)) * 4 KByte = 4 GB + 4100 Kbyte*

iv) (4 points) Consider a compact file system, called *compactfs*, tries to save as much space as possible within the inode. Thus, to point to files, it stores only a single 32-bit pointer to the first block of the file. However, blocks within compactfs store 4,092 bytes of user data and a 32-bit next field (much like a linked list), and thus can point to a subsequent block (or to NULL, indicating there is no more data). How many blocks does a file of 10KB contain?

Answer: 3

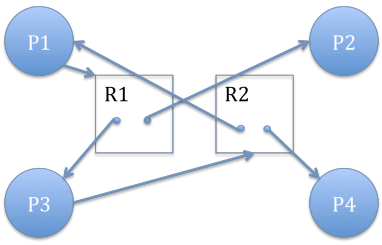


v) (4 points) What is the maximum file size for *compactfs* (assuming no other restrictions on file sizes)?

A: The maximum compactfs file size is 2^{32} x 4 KByte – 2^{32} x 4 byte = 16 TByte

5. Deadlock. Consider a system with four processes P1, P2, P3, and P4, and two resources, R1, and R2, respectively. Each resource has two instances. Furthermore: - P1 allocates an instance of R2, and requests an instance of R1; - P2 allocates an instance of R1, and doesn't need any other resource; - P3 allocates an instance of R1 and requires an instance of R2; - P4 allocates an instance of R2, and doesn't need any other resource. (**15 points, 5 points each question**)

(a) Draw the resource allocation graph



(b) Is there a cycle in the graph? If yes name it.

A: P2 and P4 are running, P1 is waiting for R1, and P2 is waiting for R2.

(c) Is the system in deadlock? If yes, explain why. If not, give a possible sequence of executions after which every process completes.

A: There is a cycle, but no deadlock.
- P2 finishes, release R1;
- P4 finishes, release R2;
- P1 acquires R1, finishes and release R1,R2;
- P3 acquires R2, finishes and release R1,R2;

6. File system reliability. Professor Harry writes the following program for grading students' projects. Per-project grades are stored in a set of input files, and the following program's goal is to compute a final course grade for each student and write it to file name.grade. **5 points**

```
main():
     remove all files ending with ".grade"
     for each student name s in alphabetical order:
             read assignment scores for student s
             calculate final grade filename = s + ".grade"
             fd = creat(filename)
             write(fd, final grade, ...)
             close(fd)
             printf("finished with %s\n", s)
```

Harry uses a laptop with a journaling file system in a mode that the journal contains both file content and the metadata. He runs the following command:

program | cat

Harry sees "finished with x" for all students with names up through "p", and then his laptop crashes. Bob reboots his laptop.

Harry thinks he may have to re-run the program for some or all students. Explain what guarantees he has about which final grades will be on disk after the restart.

Answer: If x.grade exists on the disk, then all files that precede x in alphabetical order are guaranteed to be on disk and complete. The last file might be before or after *p* and might be truncated.

7. Complete the put below using compare_and_swap so that concurrent invocations will run correctly without using locks: **10points**

```
static void put(int key, int value)
 {
    struct entry *n, **p;
    struct entry *e = malloc(sizeof(struct entry));
    e->key = key;
    e->value = value;


    for (p = &table[key%NBUCKET], n = table[key % NBUCKET]; n != 0;
                                          p = &n->next, n = n->next)
     {



            if (n->key > key) {









             }
     }



     done: return;
}
```

Answer:

```
again:
    for(p = &table...)
    {
      if(n->key > key)
      { e->next = n;
        If(compare_and_swap(p, n, e) == 0) goto again;
        else goto done;
```

```
        }
      }

      e->next = 0;

      if(bool_compare_and_swap(p, 0, e) == 0) goto again;
      done: return;
```

8. Consider the following implementation of a reader writer lock. A reader writer lock allows either multiple readers to have access to a critical section or a single writer.

```
struct rwlock
{
  sem_t *sem;
  int readers;
  int writers;
};

void rwlock_init(struct rwlock *lock)
{
   sem_init(&lock->sem, 1);
   lock->readers = 0; lock->writers = 0;
}

void readlock(struct rwlock *lock)
{
   while(1)
   {
      sem_wait(lock->sem);
      if(lock->writers == 0) { lock->readers++; break; }
      sem_post(lock->sem);
   }
}

void writelock(struct rwlock *lock)
{
    while(1)
    {
      sem_wait(lock->sem);
      if(lock->readers == 0 && lock->writers == 0) { lock->writers = 1; break; }
      sem_post(lock->sem);
    }
}
```

```
void unlock(struct rwlock *lock)
{
   sem_wait(lock->sem);
   if(lock->readers > 0) lock->readers--;
   else lock->writers--;
   sem_post(lock->sem);
}
```

(a) (7 points) What is the problem with this implementation?

Answer: When either a read or write lock is acquired, the function returns without calling sem_post(), so all future calls to sem_wait() will block forever.

(b) (8 points) Starvation is a problem where one thread is unable to acquire a resource. After fixing the problem, is starvation possible? How?

Answer: Yes. Writers can be starved, since as long as one reader remains in the critical section at all times (that is, lock->readers remains greater than or equal to one), a writer will never be able to acquire the lock.