CS 111 Final (part 2) 06/12/2017

Name _____ exam # _____

Student ID # _____

This exam is comprised of six problems, each problem being a full page of related questions. You must choose three of these problems to answer. You cannot answer more than three problems. Review all of the questions and consider what answer you would give to EACH part before deciding which three to answer. All problems have the same value.

Read each question CAREFULLY, make sure you understand EXACTLY what question is being asked and what type of answer is expected, and make sure that your answer clearly and directly responds to the asked question. Many students lose many points for answering questions other than the one I asked. If you are unsure of what a question is asking for, raise your hand and ask.

I am looking for depth of understanding and the ability to solve real problems. I want to see specific answers. One-liners and vague generalities will receive little or no credit. Superficial answers that I may have accepted previously will not be accepted on this exam.

None of these questions requires long answers, but many of the questions may require you to do a lot of thinking and sketching before you come up with a reasonable answer. Feel free to use scratch paper to organize your thoughts. If the correct part of your answer is buried under a mountain of rambling, we may not find it.

If you need more space, you can overflow onto the back page. Note (on the problem page) that you are doing this.

If you answer more than three problems, we will only grade the first three. Circle the numbers of the three problems you want us to grade:

1. (design) User-Mode Threads Package

2. (concept) Dynamic Equilibrium

3. (code) Find and Fix Race Conditions

4. (approach) Deadlock Problems

5. (concept) Copy-on-Write File Systems

6. (protocol design) A New Distributed Service

Total

1: Describe how you would implement a user-mode threads package with preemptive scheduling and mutexes on a system that supported only processes. Briefly describe the major routines in this package, the major data structures you would create, the problems you would have to solve, and how you would solve each of them.
(a) key data structures: purpose, contents, and management (allocation/dealocation)

(b) key methods: signatures, brief description of functionality, and how you would implement it.

(c) Explain how you would implement preemptive scheduling.

(d) Explain how you would implement mutex operations.

2: A dynamic equilibrium is the net result of two (or more) opposing processes that drive responses to ever changing system loads.
(a) What are the two competing forces that drive a process' dynamic working set size.

(b) Explain how these forces drive the system to respond when a process starts referencing more pages per second.

(c) Explain how these forces drive the system to respond when a process starts refrencing fewer pages per second.

(d) Explain how these forces drive the system to respond to an increase in the number of in-memory processes that are competing for memory.

(e) Explain why the above (d) response is (or is not) a constructive one.

(f) Briefly describe another (unrelated to paging) dynamic equilibrium process within an Operating System, and the competing forces that robustly drive "good" resource allocation decisions.

3: (a) Study the following code, taken from a high contention, multi-threaded application (that operates on numerous queues), and circle the actual critical sections (sensitive use/updates of shared variables).

```
struct request {
        struct request *next;           // pointer to next task in list
        long key;                       // record identifier (for sorting)
        ...                             // other info ... irrelevent to problem
};
// insert a request into a queue, keep them ordered by key
void enqueue( struct request *req, struct request **head ) {
        struct request *rp, **rpp;
        for (rpp = head; rp = *rpp; rpp = &(rp->next))
                if (rp->key >= req->key)
                        break;
        req->next = rp;                 // that one is now after us
        *rpp = req;                     // previous pointer now points to me
}
// get the next request from a queue
struct request *getnext( struct request **head ) {
        struct request *rp;
        rp = *head;
        if (rp != 0)
                *head = rp->r_next;     // next becomes new first
        return( rp );
}
// remove a request from a queue
struct request *dequeue( long findkey, struct request **head ) {
        struct request *rp, **rpp;
        for (rpp = head; rp = *rpp; rpp = &(rp->next))
                if (rp->key >= findkey)
                        break;

        if (rp->key > findkey)          // passed it without finding it
                return( (struct request *) 0 );
        *rpp = rp->next;                // previous now points past us
        return( rp );                   // return found element
}
// all of the following functions must also be Deadlock and MT-safe
submit_req( ... ) {
        ...
        enqueue( req, &active_list );
        ...
}
process_req( ... ) {
        ...
        req = getnext( &active_list );
        ...
}
// this update must be made atomically
suspend_req( key, &old_queue, &new_queue  ) {
        ...
        req = dequeue( key, &old_queue );
        enqueue( req, &new_queue );
        ...
}
```

(b) Describe the approach you will take to protecting the critical sections, briefly explain why this is the right approach, and update the above code to show how you would implement safe and correct serialization.

4a: A network lock manager provides locking services for a very wide range of distributed resources that are shared by thousands of clients. The clients are a wide range of applications running on many different operating systems, and not all of the resources they need are managed by the network lock manager. How can we ensure network locks will not contribute to deadlocks among the client applications? Justify your choice.

4b: A device driver maintains a queue of pending requests. The read and write routines add requests to the queue, and the start-up routine (which can be called either from the read routine, the write routine, or the interrupt handler) takes requests off of the queue. For request scheduling reasons, the queue is implemented as a doubly linked list, which cannot be maintained with atomic instructions. How can we protect the queue without creating potential deadlocks? Justify your choice.

4c: A massively parallel data reduction engine eliminated mutual exclusion and achieved significant scalability by dividing the data set into thousands of partitions, and assigning a single thread to perform all operations on each data partition. The data reduction process is such that most operations require multiple requests for operations in other partitions. We have seen situations where the system ran out of memory, as all of the agents were trying to allocate memory at the same time to respond to requests from other agents. How can we prevent such deadlocks? Justify your decision.

4d: A massively parallel, distributed application with thousands of concurrent threads operates on millions of mutex-protected objects. One thread only needs to lock one object at a time, but it often needs to await confirmation messages (from other threads which may have to lock other objects) before it can complete an operation. How can we prevent deadlocks between object locks and response awaiting? Justify your choice.

5: (a) What does it mean to describe a file system as Copy-on-Write?

(b) Explain how it can improve file system robustness, and how it achieves this.

(c) Explain how it could achieve space savings when files (like VM images) are cloned.

(d) Describe valuable new file system functionality it can provide, and how it achieves this.

(e) What significant new problem is created by making a file system Copy-on-Write?

(f) Briefly describe how you would implement an approach to solving that problem?

6: Amazon wants to offer distributed Key-Value Stores (KVS) as a new service to be exploited both by in-cloud Virtual Machines and WAN-remote clients. The supported KVS operations will be:

```
handle = CREATE_KVS(string name)
boolean = DESTROY_KVS(handle)
string = GET(handle, string key)
boolean = PUT(handle, string key, string value)
boolean = DELETE(handle, string key)
```

(a) It has been suggested that the service access protocol should be RESTful. State two very different major benefits of a RESTful interface, that would be important to the described service, and briefly explain why each is valuable to this service.

(b) They want to ensure that a KVS is only accessible by its owner and owner-authorized agents. They do not want to have authentication dialogs or to have to consult an authorization database before a client can access the KVS. How can a KVS manager determine whether or not a request is from an authorized agent?

(c) How would a KVS owner gain the (initial) authorization to use a particular KVS?

(d) How would the owner grant other agents the authorization to use a particular KVS?

(e) If we wanted to distinguish multiple levels of access (GET, PUT, DELETE, DESTROY), how could you extend your proposal to enable this?

(f) Providing access mediation at the KVS is very important, but requests and responses exchanged over WAN-scale links would still be subject to Man-in-the-Middle attacks. Briefly describe an extension to protect external service requests from such attacks.