University of California, Berkeley
College of Engineering
Computer Science Division — EECS

Spring 2013 John Kubiatowicz

Midterm I SOLUTIONS

March 13th, 2013

CS194-24: Advanced Operating Systems Structures and Implementation

Your Name:	
SID Number:	
Circle the letters of CS194 -24 Login	First: a b c d e f g h I j k l m n o p q r s t u v w x y z Second: a b c d e f g h I j k l m n o p q r s t u v w x y z
Discussion Section:	

General Information:
This is a **closed book** exam. You are allowed 2 pages of notes (both sides). You may use a calculator. You have 3 hours to complete as much of the exam as possible. Make sure to read all of the questions first, as some of the questions are substantially more time consuming.

Write all of your answers directly on this paper. *Make your answers as concise as possible.* On programming questions, we will be looking for performance as well as correctness, so think through your answers carefully. If there is something about the questions that you believe is open to interpretation, please ask us about it!

Problem	**Possible**	**Score**
1	16	
2	22	
3	20	
4	24	
5	18	
Total	100	

[This page left for π]

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899

Problem 1: True/False [16 pts]

Please *EXPLAIN* your answer in TWO SENTENCES OR LESS (Answers longer than this may not get credit!). Also, answers without an explanation *GET NO CREDIT.*

Problem 1a[2pts]: For Test Driven Development (TDD), the important thing is to start by writing tests that pass and to make sure that your tests never fail.

True / [✓ False]

Explain: *The important principle for RDD is the RED-GREEN-YELLOW principle: Start by writing a test that fails (RED), then fix the code so that the test passes (GREEN), and finally refactor the code while continuing to pass the test (YELLOW).*

Problem 1b[2pts]: A Zombie process is one that has come back from the dead, i.e. has be restarted after being killed once.

True / [✓ False]

Explain: *A Zombie process is a process that has completed execution but still has an entry in the process table. It is waiting for someone (typically a parent) to execute a "wait" system call to get their return value. The extended presence of a Zombie often indicates a bug in the parent.*

Problem 1c[2pts]: Threads within the same process can share data with one another by passing pointers to objects on their stacks.

[✓ True] / False

Explain: *Since all the threads are executing in the same address space, they can pass pointers to one another and share data through these pointers. Such possible sharing includes objects allocated on the stack.*

Problem 1d[2pts]: A *MicroKernel* can improve the resilience of a system against bugs in the OS.

[✓ True] / False

Explain: *Since many of the traditional services of an OS are executed at user level in a MicroKernel, the resulting system is more resilient to bugs because these services (such as a File System) can crash without crashing the whole system. Wild writes in a system service cannot affect the behavior of the core scheduler or other services because of address space protection.*

Problem 1e[2pts]: Anything that can be done with a monitor can also be done with semaphores.

[✓ True] / False

Explain: *This is true since one can implement locks and condition variables with semaphores. Locks are simple mutexes (semaphores initialized to 1), condition variables a bit more complex but can be done as well.*

Problem 1f[2pts]: **“**Hyperthreading” is a term used to describe systems with thousands of threads.

True / [✓ False]

Explain: *Hyperthreading is Intel’s name for Simultaneous Multithreading (SMT), a technique that allows multiple simultaneous threads to be directly supported by superscalar hardware. Such threads share pipeline resources and generate higher utilization of functional units than would be possible with only one thread/pipeline.*

Problem 1g[2pts]: A Lottery Scheduler can be used to implement any other scheduling algorithm by adjusting the number of tickets that each process holds.

True / [✓ False]

Explain: *It is not possible to emulate/implement a strict priority scheduler with a lottery scheduler, since every runnable task has the potential run at any time. The only way to prevent tasks from running would be to give them zero tokens,*

Problem 1h[2pts]: A *Memory Management Unit (MMU)*, is a piece of hardware that translates virtual addresses into physical addresses.

[✓ True] / False

Explain: *This is true because it is true (sorry – not much of a better “explanation”).*

Problem 2: Short Answer [22pts]

Problem 2a[2pts]: What is priority inversion and why is it an important problem? Present a priority inversion scenario in which a lower priority process can prevent a higher-priority process from running.

Priority inversion occurs when a high priority task is prevented from making progress by a lower-priority task, often because of a lock acquisition. A simple example is one in which a low-priority task acquires a spin-lock, then the high-priority task attempts to acquire the lock. The high priority task will sit and spin waiting for the lock – preventing the low-priority task from running and releasing the lock. This scenario is a deadlock that will never resolve itself. Another example occurs with blocking locks. The low-priority task acquires the lock, the high-priority task attempts to acquire the lock but is put to sleep; meanwhile, a medium-priority task runs continuously, preventing the low-priority task from running and releasing the lock.

Problem 2b[3pts]: Some operating systems do not enforce memory isolation between processes. Describe two advantages and one disadvantage of not enforcing memory isolation.

There are a number of answers to this problem. For advantages: (1) Shared-memory communication is simple – just exchange pointers, (2) Can run on much simpler processors (i.e. embedded processors without MMU), (3) Kernel is simpler and potentially less buggy, lower power consuming, etc. For disadvantages: (1) No protection between processes, (2) no protection of kernel against bugs in user programs.

Problem 2c[2pts]: What happens when an interrupt occurs? What does the interrupt controller do?

Assuming that interrupts are enabled, when an interrupt occurs, the processor saves enough state to resume execution after the interrupt (i.e. registers, PC, etc), then disables interrupts and starts executing from a special entry point in the kernel with kernel-level priviledges.

Problem 2d[2pts]: Describe one key difference between a monolithic kernel and a modular kernel and one key difference between a modular kernel and a micro kernel.

An important difference between a monolithic kernel and a modular kernel is that the modular kernel features dynamic loading of code into the kernel address space. An important difference between a modular kernel and a micro kernel is that the modular kernel still runs all of its code with kernel privileges, where as the micro kernel runs most of its services at user level.

Problem 2e[2pts]: Why is it necessary to enter Linux via a system call as opposed to a function call?

For at least two reasons. First, one must change the privilege to kernel mode in order to execute code in the kernel. This privilege change cannot happen without a system call. Second, entry into the kernel must occur only at well defined entry points for security reasons; the system call mechanism specifies entry points by number rather than by address.

Problem 2f[2pts]: The Lab 2 handout demonstrates how x264 runs slower when it is configured to run more threads than the number of physical CPUs. What are two sources of this slowdown?

Several answers are possible here. First, the scheduling overhead of switching between multiple threads causes some slowdown. Second, the synchronization delay caused by waiting for multiple threads to finish causes computational dead time. Third, algorithmic overhead occurs due to the extra work to partition the frame without the advantage of additional parallelism.

Problem 2g[3pts]: What is an ExoKernel? What functionality is placed in a LibraryOS and how does this functionality differ from the functionality typically placed into the C library?

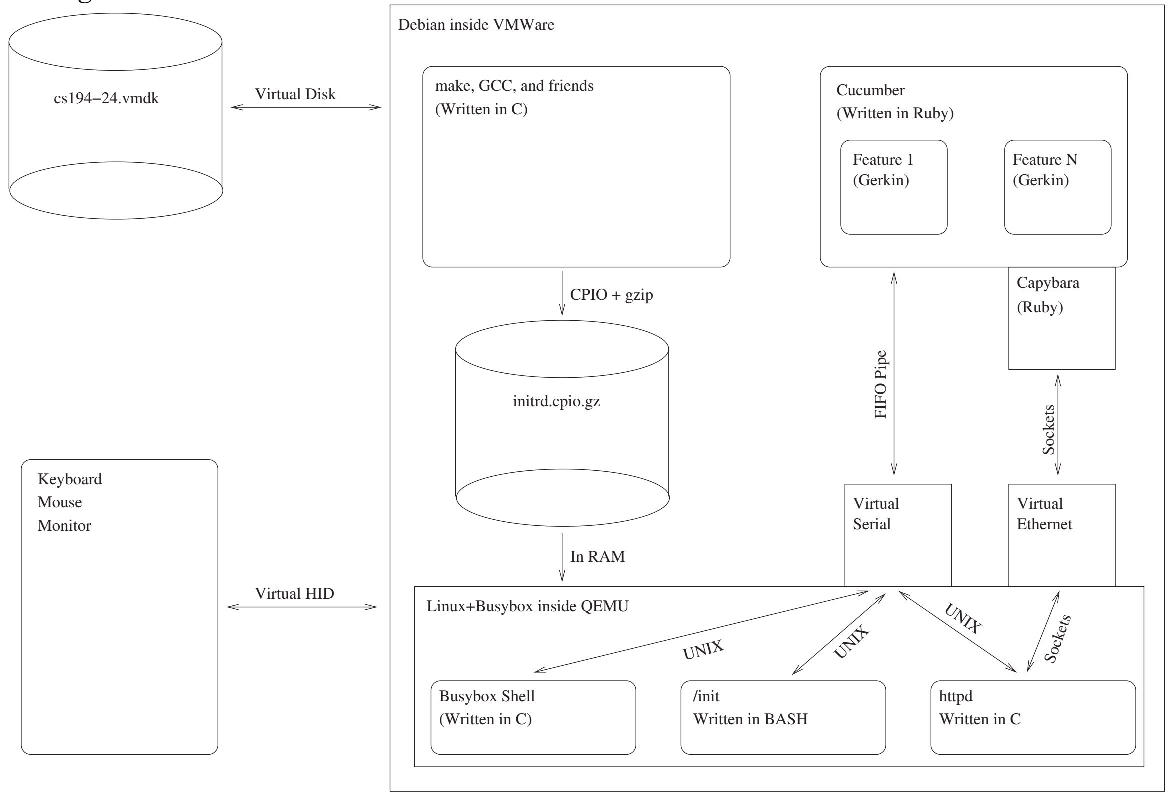
An ExoKernel is an operating systems organization that separates resource management from protection. It pushes even more functionality into user level than a microkernel. A LibraryOS contains most of the functionality normally associated with an operating system, including file systems, thread scheduling, and networking. Although the C library provides the same interface to the application developer as a LibraryOS, the C library does not implement most of the actual functionality, instead using system calls to utilize functionality from the underlying operating system.

Problem 2h[2pts]: Unlike other operating systems, Linux can utilize the same system call to generate a new process and to generate a new thread. What is this system call and how does one distinguish between different options?

The "clone()" system call can be used to generate both processes and threads. Different options are distinguished by a set of flags that determine how much of the parent's memory space and other resources (such as file descriptors) are shared by the new schedulable entity.

Problem 2j[4pts]: Cucumber is a domain-specific language (DSL) embedded in Ruby. Explain (with a diagram) how you were able to write BDD tests in Cucumber to test components of your operating system (running in a virtual machine container). Give details about which languages things are in (such as the Step definitions) and what sort of protocol runs between components.

See diagram below:



Problem 3: Lock-Free Queue [20 pts]

An object such as a queue is considered "lock-free" if multiple processes can operate on this object simultaneously without requiring the use of locks, busy-waiting, or sleeping. In this problem, we are going to construct a lock-free FIFO queue using an atomic "swap" operation. This queue needs both an `Enqueue` and `Dequeue` method.

We are going to do this in a slightly different way than normally. Rather than `Head` and `Tail` pointers, we are going to have "`PrevHead`" and `Tail` pointers. `PrevHead` will point at the last object returned from the queue. Thus, we can find the head of the queue (for dequing). Here are the basic class definitions (assuming that only one thread accesses the queue at a time). We will use Java as a slightly higher-level language for our description:

```
// Holding cell for an entry
class QueueEntry {
    QueueEntry next = null;
    Object stored;
    int taken = 0;

    QueueEntry(Object newobject) {
        stored = newobject;
    }
}

// The actual Queue (not yet lock free!)
class Queue {
    QueueEntry prevHead = new QueueEntry(null);
    QueueEntry tail = prevHead;

    void Enqueue(Object newobject)  {
        newEntry = new QueueEntry(newobject);
        tail.next = newEntry;
        tail = newEntry;
    }

    Object Dequeue() {
        QueueEntry curPrevHead = prevHead;
        QueueEntry nextEntry = curPrevHead.next;

        if (nextEntry == null) {
            return null;
        }

        prevHead = nextEntry;
        return nextEntry.stored;
    }
}
```

Problem 3a[5pts]:
Suppose that we have an atomic swap instruction. This instruction takes a local variable (register) and a memory location and swaps the contents. Although Java doesn't contain pointers, we might describe this as follows:

```
Object AtomicSwap(variable addr, Object newValue) {
    object result = *addr;    // get old object stored in addr
    *addr = newValue;         // store new object into addr
    return result;            // Return old contents of addr
}
```

Rewrite code for `Enqueue()`, using the `AtomicSwap()` operation, such that it will work for any number of simultaneous Enqueue and Dequeue operations. You should never need to busy wait. **Do not use locking (i.e. don't use a test-and-set lock).** The solution is tricky but can be done in a few lines. *Hint: during simultaneous insertions, objects may be temporarily disconnected from the queue (i.e. the set of entries reachable by* `nextEntry = nextEntry.next` *starting from* `prevEntry may not include every entry`*); but eventually the queue needs to be connected.*

Please complete the following sketch. We will be grading on conciseness; Solutions with more than 5 lines will not be accepted for full credit (you need far less):

```
void Enqueue(Object newobject)  {
    newEntry = new QueueEntry(newobject);

    // Missing code here:

    QueueEntry OldTail = AtomicSwap(tail, newEntry);
    OldTail.next = newEntry;



}
```

Problem 3b[2pts]: Provide an informal explanation of why your solution works for simultaneous enqueue operations:

We swap our new item into the tail slot with a single atomic operation. We then reconnect our new item with the previous tail at our leisure. This works under simultaneous enqueue operations because enqueuers are serialized by the swap operations. After a swap operation succeeds, the new entry is placed in the tail position. Other enqueues will connect themselves to us. Until we perform the second operation, the original queue is not connected to us, but will appear to end at the OldTail. However, when we later connect the old queue to us, dequeuers will be able to find us and anyone that has been linked to us after we started.

Problem 3c[5pts]:
Suppose that we have a compare and swap instruction. This instruction takes two local variables (registers) and a memory location and stores the second register into memory as long as the first register matches the current contents of memory. Although Java doesn't contain pointers, we might describe this as follows:

```
Boolean CAS(variable addr, Object oldValue, Object newValue) {
    object curValue = *addr;  // get old object stored in addr
    if (curValue == oldValue) {
        *addr = newValue;     // store new object into addr
        return true;          // Success!
    } else {
        return false;         // Failure!
    }
}
```

Rewrite code for Dequeue() using the CAS instruction such that it will work for any number of simultaneous threads working at once. Your solution should dequeue items in the same order that they were originally enqueued. **Again, do not use locking.** There may be short-lived busywaiting during simultaneous dequeue operations. The solution is tricky but can be done by modifying a small number of lines. *Hint: why did we save prevHead into curPrevHead?*

Please complete the following code sketch. We will be grading on conciseness: Solutions with more than 10 lines additional lines will be rejected (need many fewer!)

```
Object Dequeue() {
        // Missing code here:

        do {

        QueueEntry curPrevHead = prevHead;
        QueueEntry nextEntry = curPrevHead.next;

        if (nextEntry == null) {
            return null;
        }

        // Missing code here:


        } while (!CAS(prevHead, curPrevHead, nextEntry)



        return nextEntry.stored;
    }
```

Problem 3d[2pts]: Provide an informal explanation of why your solution for dequeue() works in the presence of simultaneous enqueue() and dequeue() operations:

The code works under simultaneous dequeue() operations because we take a snapshot of the current prevHead, compute what the next prevHead should be, then swap this new head in conditionally on the fact that no one else has changed prevHead during a dequeue() operation. This is a pretty standard pattern for lock-free behavior. As for why it works with simultaneous enqueue(), fundamentally, the enqueue() and dequeue() operations are working with different shared variables (i.e. "tail" vs "prevHead"). Either (1) there are no entries on the queue and prevHead.next is NULL. In this case, the dequeue() will return immediately and the enqueue() will proceed independently. Or (2) there are entries on the queue and one of these entries (not prevHead!) is pointed at by the tail. Thus, our advancement of prevHead during dequeue() will not eliminate the entry pointed to by the tail and thus enqueue() can proceed independently once again.

Problem 3f[4pts]: What is an MCS lock? Name two reasons why an MCS lock is better than a simple spin lock (say a Test&Test&Set lock) under high contention. When might an MCS lock be less desirable than a simple spin lock?

A Mellor-Crummy Scott lock (MCS) is a queue-based lock constructed with a lock-free queueing algorithm. The MCS lock is better than a TTS lock under high contention for several reasons (1) Each CPU spins on its own address while waiting; thus, when a lock is released, the bus traffic is minimized, and important fact when contention is high and (2) because of the queue, this MCS lock provides a fair, FIFO ordering of lock acquisition; each requester eventually gets access to the lock. An MCS lock might be less desirable than a simple spin lock when contention is low, since it has higher lock acquisition overhead (more instructions).

Problem 3g[2pts]: What are reactive synchronization algorithms? *Hint: one example could have something to do with your answer to (3f).*

Reactive synchronization algorithms switch between types of synchronization based on current system behavior. For instance, such algorithms may choose between a simple TTS and an MCS lock based on the current level of contention.

Problem 4: Scheduling [24pts]

Problem 4a[2pts]: Vanilla Linux provides 100 static priority levels for realtime processes and Kernel threads. Explain how you would use this functionality to perform RMS scheduling of realtime tasks. Would there be a limit to how many tasks could be simultaneously scheduled?

With RMS, you need a separate priority for each job. Jobs are sorted in the order of their periods, with the highest priority given to the job with the shortest period. Since periods are static, then you can perform RMS scheduling by assigning static priorities to each job, then scheduling it with the static priority levels. Since each unique period needs a separate priority, you could only schedule 100 different periods (number of priorities). Further, you need to make sure that the schedulability constraint is met [technically $\sum U_i < n\,(2^{1/n}-1)$ *].*

Problem 4b[2pts]: Explain the mechanism by which the O(1) scheduler attempted to enhance the user experience with interactive applications. Give details.

There is a set of non-realtime priorities (40 of them) which are used in a dynamic priority mode: each task has a home priority corresponding to its "nice" value. Then, the dynamic priority can vary ± 5 based on a set of heuristics intended to boost the priority of interactive apps and depress the priority of computationally bound apps. One of the heuristics, for instance, tracks the amount of time that a task sleeps and subtracts off the amount of computational time to use it to decide how to bump up or down the priority increment.

Problem 4c[2pts]: Explain how the CFS scheduler would schedule 4 CPU-bound processes with nice value +10, Targeted Latency = 20ms, and Minimum Granularity = 1ms. What is an important difference about how the O(1) scheduler would schedule the same 4 processes (give qualitative answer, do not need exact numbers for this latter question).

CFS divides up the Targeted Latency among all the scheduled tasks when they all have the same nice value. Consequently, it will give each of the 4 processes an equal chunk, i.e. 5ms each. The O(1) scheduler, on the other hand, will assign time slices based on nice value. Consequently, with a high nice value (low priority), the O(1) scheduler will define a relatively small(but identical) time slice to each task (i.e. finer than 5ms).

Problem 4d[3pts]: Explain one similarity and one difference between the behavior that users would see for processes within a scheduling group in CFS and for jobs within a CBS. How would other processes or jobs in the system affect this behavior?

Question #1: Similarity: For a given system configuration, each of these schedulers would give a fixed slice of processor time to their corresponding jobs as a whole (e.g. 10% of the total CPU time). As for a difference, CFS would try to give an equal fraction of the time to each job within its group (round-robin preemptive within the group), while CBS would run each of its jobs in FIFO order to completion (i.e. not preemptive within the group of jobs assigned to the CBS).

Question #2: With CFS, other jobs added to the system would reduce the overall CPU time given to the schedule group, approaching Minimum Granularity. With CBS, additional jobs in the system would have no effect on the CBS jobs beyond a certain point (beyond a certain point, new jobs would be rejected because the system was over subscribed); CBS would always get its requested resources.

Problem 4e[2pts]: What is two-level scheduling? (i.e. as in the Tessellation Operating system)?

Two-Level scheduling splits resource scheduling into two pieces: assigning resources (like fractions of CPU cycles or memory bandwidth) and using resources (with a custom per-application scheduler). In Tessellation, resources were assigned to Cells by a resource manager, then a custom scheduler in each Cell would use resources in an application-specific way.

Problem 4f[3pts]: The CBS paper provides the following prescription of how to handle the arrival of a new job when the CBS queue is empty:

> *When a job $J_{i,j}$ arrives and the server is idle, if $c_s \geq (d_{s,k} - r_{i,j})\ U_s$ the server generates a new deadline $d_{s,k+1} = r_{i,j} + T_s$ and c_s is recharged to the maximum value Q_s, otherwise the job is served with the last server deadline $d_{s,k}$ using the current budget.*

Here, U_s is the server utilization (Q_s/T_s). Explain the three domains of operation here (i.e. why they make sense): (1) $c_s > (d_{s,k} - r_{i,j})\ U_s$, (2) $c_s = (d_{s,k} - r_{i,j})\ U_s$, and (3) $c_s < (d_{s,k} - r_{i,j})\ U_s$

Ideally, we would like to avoid resetting the deadline too frequently, since this can give an unfair advantage to the CBS (which we are trying to avoid).

In the first case, using all of the remaining budget (c_s) immediately, without resetting the deadline, would violate the utilization constraint U_s; thus, we must reset the deadline. In the second case, using the remaining budget without resetting the deadline would exactly meet the utilization, so we could go either way; this algorithm resets the deadline to make sure that the behavior for realtime jobs is without breaks in scheduling (think about using EDF jobs which are exactly scheduled at the period). Finally, in the third case, we are easily within the utilization constraint to continue using the budget without resetting the deadline and wish to leave the deadline alone to avoid giving an unfair advantage to the CBS.

Problem 4g[2pts]: Here is a table of realtime periodic tasks with their associated arrival times, computational times/activation and periods.

Task	Arrival Time	CPU Usage per period	Period
Task A	0	1	4
Task B	0	2	5
Task C	0	1	7
Task D	0	1	8

Would it be possible to schedule these jobs with RMS? How about EDF? Explain.

Compute Total Utility: $\sum U = 1/4 + 2/5 + 1/7 + 1/8 = 0.917$

For RMS, formula converges to needing approximately total utility < .7. So, not guaranteed that it is possible to schedule. (specifically, $\sum$ *must be* $< n(2^{1/n}-1) = 4(2^{1/4}-1) = .76$*)*

For EDF, only need to make sure that total utility < 1 $\Rightarrow$ *Yes, should be possible to schedule.*

Problem 4h[8pts]: Show the scheduling order for these processes given in (4h) under 3 policies: Round-Robin (RR) with timeslice quantum = 1, Rate Monotonic Scheduling (RMS), and Earliest Deadline First (EDF). *For RR, assume that context switch overhead is 0 and that each period adds runtime to tasks still on the run-queue and adds unscheduled tasks to the back of the queue between time slots.* We filled out the first line for you. Don't forget the number of missed deadlines at bottom:

Time Slot	Newly Scheduled	RR: Q=1	RMS	EDF
0	ABCD	A	A	A
1		*B*	*B*	*B*
2		*C*	*B*	*B*
3		*D*	*C*	*C*
4	*A*	*B*	*A*	*A or D*
5	*B*	*A*	*B*	*D or A*
6		*B*	*B*	*B*
7	*C*	*B*	*C*	*B*
8	*AD*	*C*	*A*	*A*
9		*A*	*D (missed!)*	*C*
10	*B*	*D*	*B*	*B*
11		*B*	*B*	*B*
12	*A*	*B*	*A*	*A or D*
13		*A*	*D*	*D or A*
14	*C*	*C*	*C*	*C*
15	*B*	*B*	*B*	*B*
	Missed Deadlines	*None*	*Missed D=1*	*None*

Problem 5: Potpourri [18pts]

In Lab 2 you are implementing a kernel snapshotting interface that is designed to allow you to easily introspect your kernel from userspace code. In case you don't remember, the prototype for the snapshot() library routine is as follows:

```
/*
 * Generates a set of snapshots: These snapshots end up in statically
 * allocated kernel buffers, so there is a maximum number of events you can
 * ask for at once.
 *
 * events         An array of length "n" of the events to trigger on
 *
 * triggers       An array of length "n" of the trigger types
 *
 * n              The length of those arrays
 *
 * return         0 on success, -1 on failure.  This call is non-blocking
 *
 * EINVAL         The kernel cannot take "n" snapshots at once.
 */
int snapshot(enum snap_event *events, enum snap_trig *triggers, size_t n);
```

Problem 5a[2pts]: Describe on reason why we chose to use the snapshot interface given in Lab 2, as opposed to testing CBS by using the existing kernel/proc interfaces (for instance manually walking the /proc tree):

Race conditions occur when walking the proc structure: not all reads are at the same time, nor do they occur in well-defined points in the schedule. In order to check invariants, we chose the snapshot mechanism to catch a series of consistent values grabbed at well-defined points in the scheduling algorithm.

Problem 5b[4pts]: Give an example of a Cucumber feature (show code) that could utilize the snapshot interface in order to test that your scheduler was successfully performing EDF scheduling of hard realtime tasks:

When I create the following tasks

Name	*CPU*	*Deadline*
A	*1*	*2*
B	*2*	*5*

Then after one cycle, A was scheduled
And after one cycle, B was scheduled
And I create Name: C CPU: 17 Deadline: 4
And after one cycle C was scheduled

Problem 5c[3pts]: The snapshot interface was designed to be easily extensible such that it can be used to test many different types of kernel functionality. Describe how you might extend the given snapshot interface to support testing of the sub-process clone() limits you implemented in Lab 1.

Add a "process created" event, map "device" to a subtree's parent's PID, then print PID, PPID, and limits to /proc

Problem 5d[2pts]: In Lab 1, we tested subprocess clone() limits by creating a dummy application that set the limit, attempted to clone() many times, and verified that the limit was enforced. Describe one advantage that use of your extensions in (5c) would have over this previous testing methodology:

+ Ability to test real applications, since the system will work while real apps are running.
+ Statistical testing of running systems by periodically checking /proc values from snapshot

Problem 5e[2pts]: Suppose we have 2 users of a cloud system which contains 20 CPUs and 200GB of memory. Suppose that their job requirements are as follows:

User Name	**CPU Requirements**	**Memory Requirements**
User A	1 CPU	20 GB
User B	6 CPU	20 GB

How many jobs for User A and User B will be granted if we have a DRF policy? How much resource of each type will be unused with this allocation?

Renormalizing resources, we see that User A needs 5% CPU and 10% Memory per task, while User B needs 30% CPU and 10% Memory per task. Depending on whether you start with User A or User B in the iterative process, you might get one of two different results:

User A: 6 tasks, User B: 2 tasks
Or User A: 7 tasks, User B: 2 tasks

Problem 5f[2pts]: Suppose you build a device driver that receives characters via an interrupt (such as a serial port) and places them on a FIFO queue in memory for later retrieval by applications. Explain (1) why you need to have some sort of synchronization between the device driver and system call code that retrieves information from the FIFO, and (2) explain why you cannot use a semaphore for this purpose.

Because interrupts can occur at any time, the read() system call might be interrupted in the middle of updating the FIFO queue head/tail pointers. Consequently, we need synchronization between the various entities that access the FIFO queue. You cannot use a semaphore for this purpose because you are not allowed to sleep in the interrupt handler.

Problem 5g[3pts]: You are trying to find Waldo! You've managed to do all sorts of fancy image processing magic such that you've identified each person, the color of their hat, and their position on the screen. This data has been placed into the following linked list, where "struct list_head link" uses the Linux kernel's linked list implementation:

```
struct person {
  struct list_head link;
  u8 hat_r, hat_g, hat_b;
  u16 posn;
}
```

Write a method that searches a given list of people for the person whose hat color matches the given person's hat color (given as a tuple of red, green, blue). This function should return the position of the matching person, or "-1" on failure. Here is a prototype for one possibly useful macro:

```
/**
 * list_for_each_entry-iterate over list of given type
 * @pos:    the type * to use as a loop cursor.
 * @head:   the head for your list.
 * @member: the name of the list_struct within the struct.
 */
#define list_for_each_entry(pos, head, member)
```

Place your code here:

```
u16 where_is_waldo(struct person *list, struct person *to_match) {

   struct person * cur;
   list_for_each_entry(cur, list, link) {
       if ((cur->hat_r == to_match->hat_r) &&
           (cur->hat_g == to_match->hat_g) &&
           (cur->hat_b == to_match->hat_b))
          Return cur->posn;
   }

}
```

[This page intentionally left blank]

[Scratch Page: Do not put answers here!]

[Scratch Page: Do not put answers here!]