Midterm #2

CMSC 433: Programming Language Technologies and Paradigms

November 22, 2010

Name ____

Instructions

Important

- You may write **Punt** on any one question (or part of a question, if it has multiple parts) and receive 5 points or full credit on the question, *whichever is lower*.
- *Do not start until told to do so!*

Routine

- This exam has 12 pages (including this one); make sure you have them all
- You have 75 minutes to complete the exam
- The exam is worth 100 points. Allocate your time wisely: some hard questions are worth only a few points, and some easy questions are worth a lot of points.
- If you have a question, please raise your hand and wait for the instructor.
- You may use the back of the exam sheets if you need extra space.
- To be eligible for partial credit: show your work and clearly indicate your answers.
- **Write neatly** (this means you, John Toman). Credit cannot be given for illegible answers.

	Question	Points	Score
	Short Answer	25	
	Deadlock	20	
	Basic Erlang	25	
	Erlang Programming	15	
	Nonblocking Algorithms	15	
	Total	100	

1. (Short answer, 25 points)

   (a) (5 points) Name two (out of several possible) ways to reduce lock contention.

   (b) (5 points) Name one advantage and one disadvantage of Erlang's style of concurrency vs. that of Java.

   (c) Suppose we implemented a parallel maze solver that did several entire depth-first searches in parallel, but where each one followed the possible directions arising at `Choice` points in a different order; e.g., one task might go left, right, forward, while another might go forward, left, right.

   Call this strategy *parallel single-DFS.*

   Next page ⇒ ...

i. (5 pts) Summing the work done by all threads, do you think parallel-single DFS will do more work or less work than parallel BFS (breadth-first seach), e.g., as implemented in our canonical solution?

ii. (5 pts) Explain when parallel single-DFS will beat single-threaded DFS.

iii. (5 pts) Explain when parallel single-DFS will not beat single-threaded DFS.

2. (Deadlock, 20 points)

Indicate whether the following programs can deadlock (four parts, each worth 5 points, continuing onto the next page). For partial credit, give reasons for your answers. All problems will make use of the following class:

```
class Silly {
  private int x = 0;
  public synchronized int set(int y) { x = y; return y; }
  public static Runnable makeTask(Silly c1, Silly c2) {...} // see below
  public static void main(String args[]) {
    Silly c1 = new Silly();
    Silly c2 = new Silly();
    for (int i=0; i<1000; i++) {
      if (i % 2 == 0) {
        new Thread(makeTask(c1, c2)).start();
      } else {
        new Thread(makeTask(c2, c1)).start();
} } } }
```

(a)
```
public static Runnable makeTask(Silly c1, Silly c2) {
  final Silly fc1 = c1, fc2 = c2;
  return new Runnable() {
    public void run() {
      fc1.set(fc2.set(1)); }
  };
}
```

(b)
```
public static Runnable makeTask(Silly c1, Silly c2) {
  final Silly fc1 = c1, fc2 = c2;
  return new Runnable() {
    public void run() {
      synchronized (fc1) {
        synchronized (fc2) {
          fc1.set(fc2.set(1)); } } } };
}
```

(c)
```
public static Runnable makeTask(Silly c1, Silly c2) {
  final Silly fc1 = c1, fc2 = c2;
  return new Runnable() {
    public void run() {
      int hc1 = fc1.hashCode(), hc2 = fc2.hashCode();
      if (hc1 <= hc2) {
        synchronized (fc1) {
          synchronized (fc2) {
            fc1.set(fc2.set(1)); } }
     } else {
        synchronized (fc2) {
          synchronized (fc1) {
            fc1.set(fc2.set(1)); } } } } };
}
```

(d)
```
public static Runnable makeTask(Silly c1, Silly c2) {
  final Silly fc1 = c1, fc2 = c2;
  return new Runnable() {
    public void run() {
      synchronized (Silly.class) {
        synchronized (fc1) {
          synchronized (fc2) {
            fc1.set(fc2.set(1)); } } } } };
}
```

3. (Basic Erlang, 25 points)

   Look at each of the Erlang programs below, and explain what they do. Each is worth 5 points. If a program returns, indicate the value returned; if it fails with some exception, indicate the exception; or if it does not return, say so. Explain your reasoning if you hope for partial credit. You may assume all of the programs compile.

   (a) What will `go([4,3,2])` do?

   ```
   go([]) -> 1;
   go([H|T]) -> H * go(T).
   ```

   (b) What will `go([1,2],4)` do?

   ```
   go([],X) -> [X];
   go([H|T],X) -> [H|go(T,X)].
   ```

   (c) What will `go(5)` do?

   ```
   go(N) ->
     Pid = self(),
     Pid ! 2,
     receive X -> X+1 end.
   ```

Next page ⇒ ...

(d) What will `go([1,2,3])` do?

```
go(L) ->
  Pid = self(),
  Pids = lists:map(fun(N) -> spawn(fun() -> Pid ! (N-1) end) end,L),
  lists:map(fun(_) -> receive X -> X end end, Pids).
```

(e) What will `go(5)` do?

```
go(N) ->
    First = create_process_list(N, self()),
    First ! {"Hello World", 0},
    receive
      X -> X
    end.
wait(Next) ->
    receive
        {X, N} -> Next ! {X, N+2}
    end.
create_process_list(0, Next) -> Next;
create_process_list(N, Next) ->
    Last = spawn(fun() -> wait(Next) end),
    create_process_list(N-1, Last).
```

4. (Erlang programming, 15 points)

   The following code implements an atomic (mutable) integer using a server loop in the style of the sequence and semaphore examples we saw in class. The implementation currently has set and get operations. We have begun the implementation of *compare-and-swap* as the function `cas(AtomicInt,OldVal,NewVal)`; your job is to finish it. Here, `AtomicInt` is the integer, `OldVal` is its expected current value, and `NewVal` is the new value to set it to. The operation will set the integer to `NewVal` assuming the current value is `OldVal`; returns the current value.

   If you don't immediately see how to do this, I suggest you step through the test case at the bottom to see how get and set are working. Hopefully that gives you an insight as to how to do cas. You might want to sketch your answer below and then fill in your final answer on the next page.

   Next page ⇒ ...

```
make(InitVal) -> spawn(fun() -> loop(InitVal) end).

loop(CurVal) ->
    receive
        {From, Id, get} -> From ! {Id, CurVal},
                           loop(CurVal);
        {set, NewVal}   -> loop(NewVal);
        %% FILL IN to deal with CAS









    end.

get(AtomicInt) ->
    Id = make_ref(),
    AtomicInt ! {self(), Id, get},
    receive {Id, CurVal} -> CurVal end.

set(AtomicInt,NewVal) ->
    AtomicInt ! {set, NewVal}, ok.

cas(AtomicInt,OldVal,NewVal) -> %% FILL IN









% test case (should produce no match failures)
test() ->
    A = make(5),     % creates A with initial value 5
    5 = get(A),      % gets A value: 5
    set(A,6),        % sets A to 6
    6 = cas(A,5,5),  % CAS failure: passes oldval 5, but currval is 6
    5 = cas(A,6,5),  % CAS success: passes oldval 6, matches currval
    ok.
```

5. (Nonblocking algorithms, 15 points)

   Implement a (non-reentrant) spinlock in Java. Recall that a spinlock is one in which the `lock()` implementation does not block the calling thread, but continuously retries if it fails to acquire the lock. If a thread calls `unlock()` on a lock that it has not locked, the method throws an `IllegalArgumentException`.

   Your implementation will use an `AtomicReference`; its relevant API is given below. You may want to use `Thread.yield()` to voluntarily release the scheduler, and `Thread.currentThread()` to get the currently running `Thread` object.

   Next page ⇒ ...

```
public class SpinLock {
  private AtomicReference<Thread> owner =
    new AtomicReference<Thread>(null);

  public void lock() { // FILL IN









  }
  public void unlock() { // FILL IN









  }
}
```

For your reference:

```
public class AtomicReference<T> {
  T get();
  void set(T);
  boolean compareAndSet(T oldVal, T newVal);
    // sets the reference to newVal if its current value is oldVal;
    //   returns true if the operation succeeded (i.e., the reference
    //   was set to newVal), false otherwise
  ...
}
```