Midterm #2

CMSC 433: Programming Language Technologies and Paradigms

November 22, 2010

Name ____________________________________________________________

Instructions

Do not start until told to do so!

• This exam has 9 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.
• In order to be eligible for partial credit, show all of your work and clearly indicate your answers.
• Write neatly (yes, this means you, John Toman). Credit cannot be given for illegible answers.

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Answer</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Deadlock</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Basic Erlang</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Erlang Programming</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Nonblocking Algorithms</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
1. (Short answer, 25 points)

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

Answer:

The three possibilities (you just pick two of them) are (1) Reduce duration that locks are held, e.g., using smaller synchronized blocks; (2) Reduce frequency of lock requests, e.g., via lock splitting or striping; (3) Replace exclusive locks with coordination mechanisms that permit greater concurrency, e.g., Reader/Writer locks, non-blocking data structures etc.

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

Answer:

Because all data in Erlang is immutable, there can be no data races. There can be atomicity violations, but they are much harder to create. On the other hand, it can be more tedious to organize communications through message passing as compared to reading and writing shared data directly.

(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 points) 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 search), e.g., as implemented in our canonical solution?

Answer:

More work. In the BFS case, each node of the maze is examined at most once. In parallel single-DFS, each node may be visited by each task.

ii. (5 points) Parallel single-DFS will sometimes beat single-threaded DFS. Explain why.

Answer:

It will win when one of the parallel DFSs happens to find the solution faster, based on the order of choices made, than the original DFS. This solution will have to be fast enough to overcome the cost of creating and running the task, though.

iii. (5 points) Parallel single-DFS will sometimes fail to beat single-threaded DFS. Explain why.

Answer:

It could be that the single-threaded DFS happens to get lucky and finds the exit almost directly, without having to backtrack. In this case, the cost of creating extra tasks will cause the parallel single-DFS to be slower.
2. (Deadlock, 20 points)

Indicate whether the following programs can deadlock (4 parts, continues onto the next page). For partial credit, give reasons for your answers. All problems will make use of the following class:

```java
class Container {
    private Object value = null;
    public Container(Object x) { value = x; }
    public synchronized Object get() { return value; }
    public synchronized Object set(Object x) {
        Object y = value; value = x; return y;
    }
    public static Runnable makeTask(Container c1, Container c2) {...} // see below
    public static void main(String args[]) {
        ExecutorService exec = Executors.newCachedThreadPool();
        Container c1 = new Container("hi"), c2 = new Container("bye");
        for (int i=0; i<1000; i++) {
            if (i % 2 == 0) {
                exec.execute(makeTask(c1, c2));
            } else {
                exec.execute(makeTask(c2, c1));
            }
        }
        exec.shutdown();
    }
}
```

(a) `public static Runnable makeTask(Container c1, Container c2) {...} // see below`  
Answer:  
No deadlock. Only one lock is ever held at once.

(b) `public static Runnable makeTask(Container c1, Container c2) {...} // see below`  
Answer:  
Can deadlock. Sometimes fc1 will be the first Container, and sometimes the second, so its possible for tasks to be waiting on the others' container locks, in a cycle.
(c) public static Runnable makeTask(Container c1, Container c2) {
    final Container 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(fc1.get()));
                    }
                }
            } else {
                synchronized (fc2) {
                    synchronized (fc1) {
                        fc2.set(fc1.set(fc2.get()));
                    }
                }
            }
        }
    };
}

Answer:

Can deadlock, though rarely. In the case that the two containers have the same hashcode (i.e., the first if branch succeeds because of equality), then the tasks can acquire the locks in opposite orders, creating the cycle necessary for deadlock.

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

Answer:

Cannot deadlock. There’s no way for one thread to hold one lock while waiting for another: the Container.class lock is always acquired first, and once acquired, the other two locks will always be free.
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 \texttt{go([4,3,2])} do?
\begin{verbatim}
go([],) -> 1;
go([H|T]) -> H * go(T).
\end{verbatim}

\textbf{Answer:}
\begin{itemize}
  \item returns 24
\end{itemize}

(b) What will \texttt{go([1,2],1])} do? (Recall that ++ is list concatenation.)
\begin{verbatim}
go([H|T]) -> go(H) ++ go(T);
go([]) -> [],
go(X) -> [X].
\end{verbatim}

\textbf{Answer:}
\begin{itemize}
  \item returns [1,2,1]
\end{itemize}

(c) What will \texttt{go(5)} do?
\begin{verbatim}
go(N) ->
  Pid = self(),
  Pid ! 2,
  receive X -> X+1 end.
\end{verbatim}

\textbf{Answer:}
\begin{itemize}
  \item returns 3.
\end{itemize}

(d) What will \texttt{go([1,2,3])} do?
\begin{verbatim}
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).
\end{verbatim}

\textbf{Answer:}
\begin{itemize}
  \item returns [0,1,2]
\end{itemize}

(e) What will \texttt{go(5)} do?
\begin{verbatim}
go(0) -> create_process_list(0, self()),
go(N) ->
  First = create_process_list(N, self()),
  create_process_list(0, Next) -> Next;
  create_process_list(N, Next) ->
    Last = spawn(fun() -> wait(Next) end),
    create_process_list(N-1, Last).
\end{verbatim}

\textbf{Answer:}
\begin{itemize}
  \item returns {"Hello World",10}
\end{itemize}
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. Extend this implementation to also support \textit{compare-and-swap}. More precisely: implement the function \texttt{cas(AtomicInt, M, N)} where \texttt{AtomicInt} is the integer, \texttt{M} is its expected current value, and \texttt{N} is the new value to set it to. The operation will set the integer to \texttt{N} assuming the current value is \texttt{M}; returns the current value. (You might want to sketch your answer on the back of the previous page first, and then fill in your final answer here.)

\begin{verbatim}
make(N) -> spawn(fun() -> loop(N) end).

loop(N) ->
    receive
      {From, Id, get} -> From ! {Id, N},
       loop(N);
      {set, M} -> loop(M);
      %% FILL IN to deal with CAS
      {From, Id, cas, N, M} ->
        From ! {Id, N},
        loop(M);
      {From, Id, cas, _, _} ->
        From ! {Id, N},
        loop(N)
    end.

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

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

cas(AtomicInt,N,M) -> %% FILL IN
    Id = make_ref(),
    AtomicInt ! {self(), Id, cas, N, M},
    receive
      {Id, X} -> X
    end.

% test case

test() ->
    S = make(5), % sets S to 5
    5 = get(S), % gets S value: 5
    set(S,6), % sets S to 6
    6 = cas(S,5,5), % CAS failure: passes oldval 5, but currval is 6
\end{verbatim}
6 = cas(S,6,5), % CAS success: passes oldval 6, matches currval
5 = get(S), % gets S value, now 5 from successful CAS
ok.
5. (Nonblocking algorithms, 15 points)

Implement a (non-entrant) 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.

Hints: do this using an AtomicReference and its compareAndSet() operation; its API is given at the bottom of the page. You may want to use Thread.yield() to voluntarily release the scheduler, and Thread.currentThread() to get the currently running Thread object.

```java
public class SpinLock {

    public void lock() { // FILL IN
    }

    public void unlock() { // FILL IN

}

For your reference:

```java
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
    ...
}

Answer:
```
public class SpinLock {
    AtomicReference<Thread> holder = new AtomicReference<Thread>(null);
    public void lock() {
        Thread me = Thread.currentThread();
        while (true) {
            if (holder.compareAndSet(null, me))
                break;
            else
                Thread.yield();
        }
    }
    public void unlock() {
        Thread me = Thread.currentThread();
        if (holder.get() == me)
            holder.set(null);
        else
            throw new IllegalArgumentException("current thread not holder");
    }
}