CpSc 448B — Final Exam — December 10, 2009

Questions

1. (**20 points**)
   Give short definitions for each of the terms below.

   (a) Block cyclic allocation

   (b) Condition variable

   (c) Deadlock

   (d) Sequential consistency

   (e) Speed-up

2. (**20 points**)
   Consider a one-dimensional array of $N * P$ elements distributed across P processors. In particular, processor 0 holds elements $0 \ldots N-1$, and processor i holds elements $i * P$ through $(i+1) * P - 1$. Describe how to use reduce and or scan operations to find the longest sequence of consecutive 3's in the array, the length of this sequence, and the index of the first element of this sequence.

   Of course, you may define generalized reduce and scan operators in Erlang, use the versions from homework 0, or the versions from the book. Just state what you assume.

3. (**20 points**)
   Consider multipling a pair of $n \times n$ matrices on a machine with p processors. For simplicity, assume that n is a multiple of p and that p is a perfect square. Assume that we implement a parallel algorithm with p processes and that the time to send m matrix elements between two processes can be done in time $t_0 * a * m$. Assume that computing the product of an $m_1 \times m_2$ matrix with a $m_2 \times m_3$ matrix takes time $bm_1m_2m_3$. Now, consider two ways of computing $Z = X * Y$ on p processors where X and Y are both $n \times n$ matrices:

   (a) (**8 points**)
   Each process holds n/p rows of each matrix. Each process sends all of its rows for Y to every other process. Based on the rows that the processes receives, and its own rows for X, the process computes its rows for Z.

   Using the model described above, how long should it take to compute Z?

   (b) (**8 points**)
   Each process holds a $(n/\sqrt{p}) \times (n/\sqrt{p})$ block for X and Y, and will compute the corresponding block for Z. To do so, it sends its blocks for X to the $sqrtp - 1$ processes that need it, and likewise for Y. After receiving the blocks that it needs, the process computes its block of Z. Using the model described above, how long should it take to compute Z?

   (c) (**4 points**)
   Is one method always better than the other, or does the choice depend on n, p, t_0, a, and/or b? Justify your answer either by showing that one is always clearly faster, or by giving the critical values for the parameters to decide the faster algorithm.

4. (**20 points**)
Let $P0$ and $P1$ be two Erlang processes, each of which has an array, A of N integers. Each process has a variable, OtherPid which is the PID for the other process. Now, consider the following Erlang code fragment:

```
A1 = lists:sort(A),
{Even, Odd} = unshuffle(A1),
OtherPid ! {self(), 0, Odd}, receive {OtherPid, 0, OtherOdd} -> ok,
A2 = lists:merge(Even, OtherOdd),
{FirstHalf, LastHalf} = lists:split(N div 2, A2),
OtherPid ! { self(), 1,
    if (MyPid < OtherPid) -> LastHalf; true -> FirstHalf end
},
receive {OtherPid, 1, OtherHalf} -> ok,
S = if
    (MyPid < OtherPid) -> lists:merge(FirstHalf, OtherHalf);
    true -> lists:merge(LastHalf, OtherHalf)
end,
```

The unshuffle function returns two lists: Even has the even-indexed elements of lists (starting from 0, very un-Erlang), and Odd has the odd-indexed elements.

Prove that at the end of executing the code above, the process with the smaller PID has a list of the smallest N elements of the two lists in ascending order, and the process with the larger PID has the largest N elements in ascending order.

5. (**20 points** + 10 extra credit)
The module `bank.erl` included in this packet is the Erlang source code for a simple model of bank accounts. Each account has an initial balance, and then a bunch of processes perform transfers between the accounts. Each transfer has a transfer Amount which is deducted from the balance of the FromAccount and added to the balance of the ToAccount. Obviously, the *total* amount of money summed over all accounts should be constant.

I tried running the program with eight accounts, and eight activity processes where each process performed 10,000 transfers. Initially, each account had a balance of $5,000, for a total of $40,000. After performing the transfers, the total was $26,181.

(a) (**5 points**)
Explain why this program does not work as intended.

(b) (**15 points**)
A solution to this problem is to add locking behaviour to the accounts. One way to do this is to modify the account process to perform the following interactions:

receive {read, Pid, Tag}: Send {Balance, Tag} to Pid (the current behavior).

receive {write, Amount}: Set the account Balance to Amount (the current behavior).

receive exit: Terminate the account process (the current behavior).

receive {lock, Pid, Tag}: Generate a "unique" Key (my solution calls random:uniform(1000000000)) and send {Key, Tag} to Pid. Subsequent requests will block unless they include the value of Key in the request (until an unlock) is performed.

receive {read, Pid, Tag, Key}: If Key matches the current key value for this account, send {Balance, Tag} to Pid. Messages of the form {read Pid, Tag} (without a key) should be blocked until the account is unlocked.

receive {write, Pid, Tag, Key}: If Key matches the current key value for this account, set the account Balance to Amount. Messages of the form {write Amount} (without a key) should be blocked until the account is unlocked.

receive {unlock, Key}: If Key matches the current key value for this account, go back to the original (unlocked) behaviour.

Write an account function that implements these behaviours.
Hint: My solution has an account function that comes in both a one-argument and a two-argument version.

(c) (**5 points, extra credit**)
I implemented a solution to part (b), and modified the get and transfer functions as shown in module `bank2.erl` (included in this packet). The code deadlocked. Explain why?

(d) (**5 points, extra credit**)
Modify the transfer function from module `bankd2.erl` to correct the deadlock problem.

bank.erl (page 1 of 2)

```
-module(bank).
-export([main/0, main/1, main/8]).
-export([account/1, transfer/3, activity/5, next/1, busy/6]).

account(Balance) ->  % keep track of a bank account's balance
  receive
    {read, Pid, Tag} ->
      Pid ! {Tag, Balance},
      account(Balance);
    {write, Amount} -> account(Amount);
    exit -> ok
  end.

get(Pid, What, Tag) ->  % get a value from an account
  Pid ! {What, self(), Tag},
  receive {Tag, Value} -> Value end.

% transfer money between accounts
transfer(FromAccount, ToAccount, Amount) when FromAccount /= ToAccount ->
  FromBalance = get(FromAccount, read, from_acct),
  ToBalance = get(ToAccount, read, to_acct),
  if
    (FromBalance >= Amount) ->
      FromAccount ! { write, FromBalance - Amount },
      ToAccount   ! { write, ToBalance + Amount },
      ok;
    true -> insufficient_funds
  end;

transfer(FromAccount, ToAccount, _Amount) when FromAccount == ToAccount ->
  ok.
% Perform a bunch of transfers between accounts.
activity(0, _Accounts, _R1, _R2, _R3) -> ok;
activity(N, Accounts, {R1a, R1b}, {R2a, R2b}, {R3a, R3b}) ->
  FromAcct = lists:nth(hd(R1a), Accounts),
  ToAcct = lists:nth(hd(R2a), Accounts),
  Amount = hd(R3a),
  transfer(FromAcct, ToAcct, Amount),
  activity(N-1, Accounts,
           next({R1a, R1b}), next({R2a, R2b}), next({R3a, R3b})).

% sweep back and forth over a list of random values
next({[H | T], X}) when T /= [] -> {T, [H | X]};
next({[E], X}) -> {X, [E]}.
```

bank.erl (page 2 of 2)

```
% Perform N transfers and then tell ParentPid that we're done.
%   Accounts is a list of Pids for accounts.
%   R1, R2, and R3 are tuples of list-pairs for randomizing the actions
busy(N, Accounts, R1, R2, R3, ParentPid) ->
  receive go -> ok end, % wait for start signal
  activity(N, Accounts, R1, R2, R3),
  ParentPid ! done.
%  A simple test routine
main(Naccounts, Nprocs, Ntransactions, N1, N2, N3, Balance0, MaxTransfer) ->
  % create accounts
  AccountPids =
    [ spawn(fun() -> account(Balance0) end) || _ <- lists:seq(1, Naccounts)],

  % create random lists of account indices and transfer amounts
  X1 = [ random:uniform(Naccounts) || _ <- lists:seq(1, N1) ],
  X2 = [ random:uniform(Naccounts) || _ <- lists:seq(1, N2) ],
  X3 = [ random:uniform(MaxTransfer) || _ <- lists:seq(1, N3) ],

  % create pids for doing transfers
  MyPid = self(),
  BusyPids = [
        spawn(fun() ->
          R1 = lists:split(random:uniform(N1-1), X1),
          R2 = lists:split(random:uniform(N2-1), X2),
          R3 = lists:split(random:uniform(N3-1), X3),
          busy(Ntransactions, AccountPids, R1, R2, R3, MyPid)
        end)
    ||  _ <- lists:seq(1, Nprocs)
  ],

  % start the frenetic activity
  [ Pid ! go || Pid <- BusyPids ],

  % wait for the BusyPids to finish
  [ receive done -> ok end || _ <- BusyPids ],

  % check the account balances
  GetBalance = fun(Acct) ->
    Acct ! {read, MyPid, {Acct, finalBalance}},
    receive {{Acct, finalBalance}, B} -> B end
  end,
  Balances = [ GetBalance(Acct) || Acct <- AccountPids ],
  io:format("Initial total = ~w, final total = ~w~n",
    [ Naccounts*Balance0, lists:sum(Balances)]),

  [ Pid ! exit || Pid <- AccountPids ], % clean-up
  Balances.

main(Nproc) -> % test case using default parameters
  main(8, Nproc, 10000, 117, 257, 83, 5000, 100).

main() -> main(8).
```

bank2.erl (page 2 of 2)

```
-module(bank2).

% stuff omitted that is either like bank.erl or is part of the solution
% to a problem for this exam.

get(Pid, What, Tag) ->
  Pid ! {What, self(), Tag},
  receive {Tag, Value} -> Value end.

get(Pid, What, Tag, Key) ->
  Pid ! {What, self(), Tag, Key},
  receive {Tag, Value} -> Value end.


transfer(FromAccount, ToAccount, Amount) when FromAccount /= ToAccount ->
  FromKey = get(FromAccount, lock, from_acct),
  ToKey = get(ToAccount, lock, to_acct);
  FromBalance = get(FromAccount, read, from_acct, FromKey),
  ToBalance = get(ToAccount, read, to_acct, ToKey),
  Result = if
    (FromBalance >= Amount) ->
      FromAccount ! { write, FromBalance - Amount, FromKey },
      ToAccount   ! { write, ToBalance + Amount, ToKey },
      ok;
    true -> insufficient_funds
  end,
  FromAccount ! unlock,
  ToAccount   ! unlock,
  Result;

transfer(FromAccount, ToAccount, _Amount) when FromAccount == ToAccount ->
  ok.

% more stuff omitted as described above
```

Some Erlang documentation

```
merge(List1, List2) -> List3

    Types:
    List1 = List2 = List3 = [term()]

    Returns the sorted list formed by merging List1 and List2. Both List1
    and List2 must be sorted prior to evaluating this function. When two
    elements compare equal, the element from List1 is picked before the
    element from List2.


nth(N, List) -> Elem
  Types:
  N = 1..length(List)
  List = [term()]
  Elem = term()

  Returns the Nth element of List. For example:

  > lists:nth(3, [a, b, c, d, e]).
  c


sort(List1) -> List2
    Types:
    List1 = List2 = [term()]

    Returns a list containing the sorted elements of List1.

split(N, List1) -> {List2, List3}
    Types:
    N = 0..length(List1)
    List1 = List2 = List3 = [term()]

    Splits List1 into List2 and List3. List2 contains the first N
    elements and List3 the rest of the elements (the Nth tail).
```

```
seq(From, To) -> Seq
seq(From, To, Incr) -> Seq
    Types:
    From = To = Incr = int()
    Seq = [int()]

    Returns a sequence of integers which starts with From and contains
    the successive results of adding Incr to the previous element, until
    To has been reached or passed (in the latter case, To is not an
    element of the sequence). Incr defaults to 1.

    Failure: If To<From-Incr and Incr is positive, or if To>From-Incr
    and Incr is negative, or if Incr==0 and From/=To.

    The following equalities hold for all sequences:

    length(lists:seq(From, To)) == To-From+1
    length(lists:seq(From, To, Incr)) == (To-From+Incr) div Incr

    Examples:

    > lists:seq(1, 10).
    [1,2,3,4,5,6,7,8,9,10]
    > lists:seq(1, 20, 3).
    [1,4,7,10,13,16,19]
    > lists:seq(1, 0, 1).
    []
    > lists:seq(10, 6, 4).
    []
    > lists:seq(1, 1, 0).
    [1]
```