CS 201 Section 1 Lab 2 – due 6pm, Wednesday, February 2, 2000

Counting Queens

The 8-queens problem is a well-known combinatorial search problem: find a way to place 8 queens on a standard chessboard so that none can capture any other. More generally, and for those who don't play chess, this means: given an n-by-n square grid, find n grid locations such that no two are in the same row, column, or diagonal.

Here are two solutions for $n = 4$:

```
3 _ _ Q _   3 _ Q _ _
2 Q _ _ _   2 _ _ _ Q
1 _ _ _ Q   1 Q _ _ _
0 _ Q _ _   0 _ _ Q _
  0 1 2 3     0 1 2 3
```

In fact, these are the *only* possible solutions for $n = 4$. Note that they are actually mirror images of each other (with respect to the X or Y axis or to either diagonal), but we still count them as distinct.

There are several variants to the problem: we can ask for one solution, all solutions, or the number of solutions. To find a single solution, the usual approach is to place queens one at a time until either a solution is reached or no legal position remains. In the former case, we record or print out the solution and terminate; in the latter case, we *backtrack*, change the position of the last queen we placed, and try again. To display or count all solutions, we simply backtrack instead of terminating after each solution is found. The number of possible positions grows exponentially with n, and the problem rapidly becomes very difficult to solve.

Here are some values for the number of solutions for different board sizes.

n	Solutions
3	0
4	2
5	10
6	4
7	40
8	92
9	352
10	724
11	2680
12	14200
13	73712
14	365596

Your task in this assignment is to write a SPARC assembly language function, callable from C, that, given input parameter n, computes and returns the *number* of n-queens solutions using the algorithm detailed below. You may assume that $n < 32$. In fact, you shouldn't try to compute the answer for $n \geq 15$, since it will take too long and consume too many machine cycles.

Here's the required C interface for your function:

```
int queens(int size);
```

A simple driver for this function is:

```
#include <stdlib.h>

int queens(int);

main (int argc, char*argv[]) {
  int size;
  if (argc != 2) {
    printf("usage: driver size\n");
    exit(1);
  };
  size = atoi(argv[1]);
  printf ("Size = %d  Solution count = %d\n", size, queens(size));
}
```

The algorithm is based on a few simple ideas. We label rows and columns from 0 to $n-1$. First, since it is obvious that each solution must place one queen in each column, we build a solution column by column. (Of course, we could just as well build by rows, but we arbitrarily choose columns. In any case, rows and columns are completely interchangeable anyhow, since we can always rotate the board by 90 degrees.) Second, we keep constant track of a set of "bad" row positions, and never try to place queens in them; the set of bad positions grows larger as we move to higher columns.

Here is the pseudo-code for the algorithm. Note that `nat` is the type of natural numbers $\{0, 1, 2, \ldots\}$ and `natset` is the type of sets of natural numbers.

```
global nat size;
global nat solutions;

void try_column(nat column,
                natset bad_rows,
                natset bad_up_diags,
                natset bad_down_diags) {
  if (column ≥ size)
    solutions++
  else {
    natset bad = bad_rows ∪ bad_up_diags ∪ bad_down_diags;
    nat candidate = 0;
    while (true) {
      while (candidate ∈ bad)
        candidate++;
      if (candidate ≥ size)
        return;
      else {
        try_column(column+1,
                   bad_rows ∪ { candidate },
                   { r+1 | r ∈ (bad_updiags ∪ {candidate}), r < size-1 },
                   { r-1 | r ∈ (bad_downdiags ∪ {candidate}), r > 0 }
               );
        candidate++;
      }
  }
}

nat queens (nat size0) {
   size = size0;
   solutions = 0;
   try_column(0,∅,∅,∅);
   return solutions;
}
```

This is a recursive algorithm, with one level of recursion corresponding to each column, and a non-recursive outer-level function to start things off. At each level, we pass three sets containing the row numbers of positions that are invalid for this column due to the placements in earlier columns. Set `bad_rows` describes positions that are in the same row as a previous queen; `bad_up_diags` describes positions are in the same up-diagonal as a previous queen; `bad_down_diags` describe positions that are in the same down-diagonal as a previous queen. Holding these three sets separately allows them to be computed incrementally at each recursive call.

Use unsigned integers to represent the `nat` type and bitmaps to represent the `natset` type. Because of the restriction on the size of n, each set will fit nicely into one SPARC word.

You will also find it convenient to represent the `candidate` value as a singleton set. The set computations at the recursive call can be performed *very* efficiently using this representation.

Some advice and warnings:

- Test using a small number of queens (e.g., 8), which should take less than one second to compute.
- Be kind to your fellow users. Don't let your tests run more than 30 seconds. Make sure you know how to terminate your program using CTRL/C!
- You can store the global variables `size` and `solution` in the `%g1` and `%g2` variables. But you should save and restore the original value of `%g2` on entry and exit from the top-level `queens` function.
- You will probably want to use `gdb`, the GNU debugger, to help debug your code. More details are in a separate handout.

To submit this lab, you need to send just one file, `queens.s`, containing all your assembly code. We'll test it using the driver above. Mail it as the body of an email message to `cs201-1acc`, e.g.,

```
mail -s "lab2" cs201-1acc
```

Your submission will be built by using the command

```
cc -o queenstest driver.c queens.s
```

and then tested by running `queenstest`