"More than the act of testing, the act of designing tests is one of the best bug preventers known. The thinking that must be done to create a useful test can discover and eliminate bugs before they are coded - indeed, test-design thinking can discover and eliminate bugs at every stage in the creation of software, from conception to specification, to design, coding and the rest." – B. Bezier

"The speed of a non-working program is irrelevant." – S. Heller (in "Efficient C/C++ Programming")

Learning Objectives

1. Assembling larger programs from components
2. Concurrency
3. Creative problem solving

Work that needs to be handed in (via SVN)

1. `spimbot.s`, your SPIMbot tournament entry,
2. `partners.txt`, a list of your 1 or 2 contributor's NETIDs,
3. `writeup.txt`, a few paragraphs (in ASCII) that describe your strategy and any interesting optimizations that you implemented, and
4. `teamname`, a name under which your SPIMbot will compete. Team names must be 40 characters or less and should be able to be easily pronounced. Any team names deemed inappropriate are subject to sanitization.

Guidelines

- **You must do this assignment in groups of 2 or 3 people.** Otherwise, we'll have too many programs for the competition.
- You'll want to use `/class/cs232/Linux/bin/QtSpimbot`. See Lab 9 for directions on running QtSpimbot.
- Use any MIPS instructions or pseudo-instructions you want. In fact, anything that runs is fair game (i.e., you are not required to observe calling conventions, but remember the calling conventions are there to help you avoid bugs). Furthermore, you are welcome to exploit any bugs in SPIMbot or knowledge of its algorithms (the full source is provided in the _shared directory), as long as you let me know after the contest what you did.
- There is no line; we will execute your code as is.
- We will not try to break your code; we will compete it against the other students.
- We've provided solution code for Labs 7-9. You are free to use any of this code in your SPIMbot contest implementation.
- The contest will be run on the EWS linux machines; so those machines should be considered to be the final word on correctness. Be sure to test your code on those machines.

Problem Statement

In this assignment you are to produce a program for SPIMbot that will allow it to collect tokens as fast as possible. On Wednesday, 12/12, we will have a double elimination tournament to see which program performs the best.

In each round of the tournament, we will compete two robots, to see which can pick up more of the tokens first. Each robot will be placed randomly on the map, as will the tokens. Unlike in Lab 9, you will not be initially provided the location of the tokens. Instead, you will have to use an I/O device, called the *scanner*, to determine their location.

To perform a scan, you specify the center — (X,Y) coordinates — and a radius for an area to be scanned. These values are written into the memory-mapped I/O addresses `SCAN_X`, `SCAN_Y`, and `SCAN_RADIUS`, respectively. To initiate the scan, the memory-mapped I/O address `SCAN_ADDRESS` must be written. The value written to this location should be a memory address where the scanner can write its results. The scanner requires that 16kB (16384 bytes) be allocated for its results; large chunks of space can be reserved in the data segment by using the `.space` directive (see Appendix A of your book). The scanner takes time to perform its work; in fact the time taken is proportional to the area scanned. When it completes its work, it writes its results to the provided memory address, and raises a `SCAN` interrupt, provided it has been enabled.

Once a scan has completed, its results need to be decoded. The scanner returns an array of 15 doubly linked lists (using the `list_t` structure defined in Lab 8). Since each `list_t` structure is 8 bytes large, `SCAN_ADDRESS` points to the first list object, `SCAN_ADDRESS`+8 points to the second, and `SCAN_ADDRESS`+112 points to the fifteenth. Each of these lists has 32 nodes; the structure of each of these nodes is similar to the `node_t` structures from Lab 8, with one extra field:

```
typedef struct node_s {
  int data;             // field for sorting
  struct node_s * prev;
  struct node_s * next;
  int value;            // field for collapsing to get coordinates
} node_t;
```

To extract the token locations, each of these lists has to be sorted (using the `data` field as the key, and then `value` fields of the resulting list have to be interpreted as a boolean and *collapsed* into a single 32-bit integer. Because the `value` field has been added to the end, the sorting code from Lab 8 should work without modification. The required "collapsing" process is the same one as was necessary in Lab 7, except here we are performing it on a list, instead of an array (as follows):

```
unsigned compact(list_t *list) {
  unsigned ret_val = 0;         /* return value */
  node_t *trav;                 /* list traversal pointer */
  unsigned int mask = 1 << 31;  /* word size = 32 bits */
  for (trav = list->head ; trav != NULL ; trav = trav->next) {
    if (trav->value != 0)       /* non-zero = TRUE */ {
      ret_val |= mask;          /* set bit within word */
    } else {
      ret_val &= ~mask;         /* clear bit within word */
    }
    mask = mask >> 1;           /* adjust mask */
  }
}
```

The resulting 32-bit number should be interpreted as two 16-bit (unsigned) integers that specify the X and Y locations of a potential token; the 16 most-significant bits of the number indicate the X coordinate and the 16 least-significant bits indicate the Y coordinate. The token location is only valid if both the X and Y coordinates are on the map (between 0 and 300, inclusive), as shown in the following code fragment:

```
unsigned result = sort_and_extract(...);  // produce 32-bit number
unsigned X_coord = result >> 16;
unsigned Y_coord = result & 0xffff;
if ((X_coord <= 300) && (Y_coord <= 300)) {
   // add token to
}
```

You might find that giving SPIMbot the `-debug` parameter will be useful for your debugging; it prints out the coordinates that are packed into lists and scrambled. You can compare your "sorted and extracted" values to these. Note: you can use the `PRINT_INT` memory-mapped I/O address to print out the values you generated.

As tokens are found, SPIMbot should be driven to collect them. The code you wrote for Lab 9 enables you to drive SPIMbot in parallel with finding more tokens.

The contest runs until all 15 tokens have been collected and the winner will be the robot that has collected the most tokens. Robots that consistently fail to collect all of the tokens when run in isolation will not be entered in the contest.

Strategy

In class, we'll discuss the merits of the iterative design process. You are strongly encouraged to, first, get a simple implementation working and then focus on optimizations. Specifically, a simple implementation would:

- request a scan that covers the entire map (center at (150, 150), radius sqrt(150*150 + 150*150)).
- receives scan interrupt and processes the scan to extract the token locations.
- sequentially drives to pick up the tokens.

In Labs 7 through 9, you have already written much of the code necessary to assemble such a contest entry. Such an entry that can consistently collect all of the tokens when playing by itself will receive 60 out of 100 points.

To improve your chances of winning the competition, you can optimize your SPIMbot program. While there are many ways that this can be done, we provide a few suggestions.

- Note that there are three things that take time: 1) scanning the map, 2) sorting and extracting the token locations, and 3) driving to pick up the tokens. Also, note that these activities can be performed concurrently, because they use different resources: scanning uses the scanner, sorting uses the CPU, and driving uses SPIMbot's wheels. You can exploit this concurrency to reduce the time it takes to collect all of the tokens, by performing multiple smaller scans and scanning one region while sorting the results from a second and driving to collect the tokens found in a third. This process is the software corollary of pipelining, which we studied for Lab 10.
- Rather than traveling the Manhattan, or rectilinear, distance (moving exclusively in the X or Y direction at a time), you could travel the Euclidean distance (the shortest distance between two points is a straight line). We've provided code that uses the MIPS floating point unit to compute the angle to drive given two (X,Y) locations. The code can be found at the following URL:
  - `http://www.cs.uiuc.edu/class/sp06/cs232/euclidean.s`