Exam 2 (solutions)

April 25th, 2005

Work alone. Do not use any notes or books. You have approximately 60 minutes to complete this exam. There are 4 problems worth 20 points each for a total of 80 points.

Please write your answers on the exam. More paper is available if you need it. Please put your name at the top of the first page.

1 A faulty program (20 points)

The following program contains three lines each of which attempts to access a memory location that is likely to be invalid, because it dereferences a null pointer, a pointer that has not been properly initialized, or a pointer that goes beyond the bounds of an array or allocated block.

Identify these faulty lines by their line numbers and write the output produced by the program if all three faulty lines are removed.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int
5   main(int argc, char **argv)
6   {
7       int *a;
8       int *b;
9       int *c;
10      int i;
11
12      *a = malloc(sizeof(int));
13      b = malloc(sizeof(int) * 2);
14      c = malloc(sizeof(int) * 3);
15
16      for(i = 0; i < 2; i++) {
17          b[i] = i;
18          c[i] = i+5;
19      }
20
21      b[2] = 12;
22      c[2] = 15;
23
24      printf("%d\n", &b[1] - b);
25      printf("%d\n", b[*c] - c[*b]);
26
27      return 0;
28  }
```

Solution:

The bad lines are 12 (`*a` points nowhere), 21 (`b[2]` is past the end of the block), and 25 (`b[*c]` is `b[5]`, also past the end of the block). The output is `1`. Note that C pointer subtraction is in units of the size of the target type, for consistency with C pointer addition.

2 A stack (20 points)

Here is a partial implementation of a stack, including a function `stack_create` for creating a new, empty stack and a function `stack_push` for adding a new value to an existing stack.

```
#include <stdlib.h>

struct stack { int value; struct stack *next; };

struct stack *stack_create(void) {
    struct stack *s;
    s = malloc(sizeof(*s));
    s->next = 0;
    return s;
}

void stack_push(struct stack *s, int value) {
    struct stack *s2;
    s2 = malloc(sizeof(*s->next));
    s2->value = value;
    s2->next = s->next;
    s->next = s2;
}
```

What is missing is a function `stack_pop` that takes a `struct stack *` as its only argument and removes and returns the most recently pushed `int` that has not already been popped. Write this function. You may assume that the user will never call `stack_pop` when the stack is empty. Your function should free any allocated storage that is no longer needed.

Solution:

```
int stack_pop(struct stack *s)
{
    struct stack *s2;
    int value;

    /* removal is the reverse of installation */
    s2 = s->next;
    s->next = s2->next;
    value = s2->value;
    free(s2);

    return value;
}
```

3 A wide tree (20 points)

Consider a tree made up of nodes declared as follows:

```
struct node {
    int n;               /* number of children, 0 if none */
    struct node **kids;  /* array of pointers to children */
};
```

where a node `r` with `r->n` children has pointers to them in `r->kids[0]`, `r->kids[1]`, ..., `r->kids[r->n-1]`. *Clarification added during the exam:* You may assume that all entries in `r->kids` are non-null.

Write a function that takes a pointer to the root of such a tree and returns the total number of nodes in the tree (including the root). Your function should run in time linear in the number of nodes. You may assume that the depth of the tree is small enough that stack size limits will not be an issue. We have written the header of your function for you:

```
int tree_size(struct node *root)
```

Solution:

```
{
    int i;
    int count;

    count = 1;
    for(i = 0; i < root->n; i++) {
        count += tree_size(root->kids[i]);
    }

    return count;
}
```

4 Two searches (20 points)

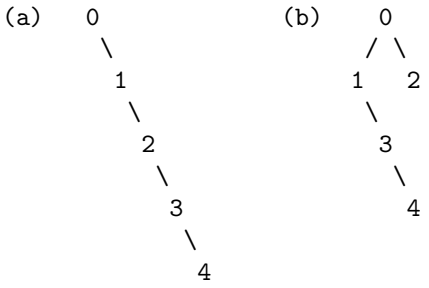
Below is a table listing the edges of a directed graph in adjacency-list form; for each line $u : v_1, v_2, \ldots v_k$ there are edges $(u, v_1), (u, v_2), \ldots (u, v_k)$.

Draw a picture of (a) a depth-first search tree starting at node 0; (b) a breadth-first search tree starting at node 0. In constructing your trees, you may choose to consider the edges leaving each node in whatever order you like.

```
0: 1 2
1: 2 3
2: 0 3
3: 1 4
4: 1 2
```

Solution:

In exciting ASCII art:



(Other possibilities exist if you process the outgoing edges in a different order.)