Assignment 2b: Memory

Assignment written by Julie Zelenski.

I had previously intended to distribute Assignment 2 in three installments, but in the interest of time and stress reduction, I've pruned previous HW2b and Hw2c down to just one installment, and here you have it.

You are expected to work through the HW2 problems, at your own pace and to your own desired level of understanding, working in groups if you like. HW2 will not be handed in or graded in order to give a bit of break from the infamous CS107 workload. Solutions will be distributed before the midterm so you can prepare for the exam.

Problem 6: Performance lab

Consider the following function that will exchange the values of two integers:

```
void SwapInt(int *a, int *b)
{
   int tmp;

   tmp = *a;
   *a = *b;
   *b = tmp;
}
```

We could create a whole family of such functions, `SwapFloat`, `SwapChar`, etc. each fixed to a particular type, or we could go for the more general `SwapAny` function that is capable of swapping any type of values, given the additional information about the size:

```
void SwapAny(void *a, void *b, int size)
{
   void *tmp = malloc(size);
   assert(tmp != NULL);
   memcpy(tmp, a, size);
   memcpy(a, b, size);
   memcpy(b, tmp, size);
   free(tmp);
}
```

As we discussed in class, the simple fixed-type function is appealing because it compiles to a short and straightforward set of machine instructions, but having compiled to that particular stream of instructions means it is not very flexible. The more general version can do it all, but at the cost of making several function calls and using dynamic memory for the temporary storage. Let's find out how that overhead adds up. In `/usr/class/cs107/assignments/hw2/swap.c`, there is a simple program with a loop that does a lot of iterations, calling either `SwapInt` or `SwapAny`. The code is commented and should be fairly self-explanatory.

a) Compile our program and run it a couple of times for each of the two test cases, and make a chart which records on average how much time each takes. On a multi-tasking machine, using your watch to time a program isn't really fair or accurate. Instead use the `/usr/bin/time` utility. This will run a command and report how long it takes both in "real" time and "user" time.[1] For example, to run "myprogram" and find out how much time it takes, use this:

```
% /usr/bin/time myprogram
```

How does the performance of hard-wired `SwapInt` compare to the generic version `SwapAny`?

b) Now try to figure out what is contributing more to the slowdown: is it the cost of passing the third parameter? the use of the heap instead of stack for storage? the calls to `memcpy` instead of direct load and stores to memory? Start with the basic `SwapInt` function and construct three different experiments that include just one piece of the extra overhead. Run the time trials on each a few times and report the result. What percentage of the overhead is due to the extra parameter? the use of the heap? the calls to memcpy?

Problem 7: Identical Outputs

As has been stressed in lecture, type information is used at compile-time to make decisions about how many bytes of data to load/store and calculating offsets and the like, but all that is left at runtime is a sequence of instructions and data laid out in memory for it to operate on. Looking at the generated code tells you little about what a given 4 bytes is: an `int`? a `struct fraction *`? a sequence of 4 characters? It is possible for many different C functions to compile to the same output— the exact same sequence of CPU instructions is appropriate if the functions access the same memory locations in the same pattern. For example, look at this set of machine instructions:

```
R2 = Mem[R1]
R3 = R2 + 4
R4 = Mem[R1 + 4]
Mem[R4] = R3
```

a) Assuming that R1 holds the address of the first local variable (and others follow at higher addresses), build a C snippet with variables of only pointer type (any type of pointer okay) that will compile to the above sequence of machine instructions. The code should not have any typecasts.

b) Construct another function with just one local variable involving only integer type (array or structs of integers also okay) that compiles to the same sequence of machine instructions. You can use typecasts if necessary.

---

[1] You might ask, what is the difference between all the times it reports? "Real" time is elapsed time, as if you were counting by looking at your watch. "User" and "system" time refer to actual time spent by this process on the CPU, which is more interesting to us. The "user" portion is the code that was executed in normal mode, the "system" portion identifies code executed on your behalf by the system (such as when you ask the OS to do some privileged thing such as an open a file). For our purposes, we are interested in "user" time, so consider only that in your answers.

In gdb, the `disassemble` command lists the generated machine code for any compiled function. For example, set a breakpoint on `main`, run your program, and when it stops at `main`, try this:

```
(gdb) disassemble strchr
```

Voila, more Sparc assembly than you ever wanted to see! You'd be surprised how much you can kind of poke your way through and understand since you have some basic idea of what a RISC assembly language looks like. For example, `ld` is the load instructions, `st` is store, the `b`, `ble`, `bg` etc., instructions are branches, etc.

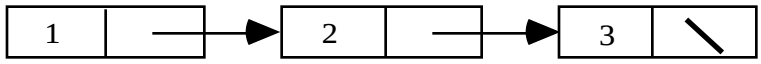
In the `swap.c` program used in the previous problem, there are a couple of other variants of the Swap function for different types (`SwapFloat`, `SwapShort`, etc.). Compile this file normally and start `gdb` on it so you can use the `disassemble` command to look at the generated machine code.

c) First off, consider the two swap functions that work on values of the same size (floats and ints are both 4 bytes). Compare the disassembly from the two functions and identify the differences in the two sequences of machine. Now consider two functions that work on different-sized values (such as ints and chars), what differences do you find in their instruction sequences?

c) Now compile the swap program with optimization, by passing the -O2 flag to gcc when compiling. (If you're curious about the types of optimizations this enables, see the gcc man page.) Re-run gdb on the optimized version and use the disassembly command to look at the functions again. Count the number of lines in the unoptimized and optimized version. Just how good is the optimizing compiler? What percentage of the instructions was it able to eliminate?

Problem 8: Find the linked list (an old CS107 exam question)

You are debugging a program. You suspect that the program has allocated a linked list containing the sequence of numbers (1,2,3), but you're not sure. If the list is present, the three elements will use the following type definition and will be allocated and set up in the normal way with calls to `malloc`.




```
struct list {
    int data;
    struct list *next;
};
```

Write a function that looks through the heap to see if a (1,2,3) list is present anywhere. Assume that the function takes no inputs but returns true if the list is present and false otherwise. You can assume that `HEAP_START` and `HEAP_SIZE` are #defined for you. Your

function should not dereference any invalid pointers. Do not worry about alignment restrictions or rounding up allocations to some larger value.

Problem 9: Experimenting with malloc and Purify

There are requirements that any `malloc` implementation must conform to such as returning a pointer to space at least as large as asked for, returning `NULL` on failure, allowing previous allocated space to be reclaimed with `free`, etc. However, there are many behaviors that are not specified, such as what happens when you free something twice, what happens if you used a freed pointer, etc., which leaves the opportunity for the `malloc` implementor to make choices.

Many `malloc` implementations tradeoff safety for efficiency, figuring that the allocation functions are so heavily used that it is of utmost importance that they be as streamlined as possible. Checking for errant requests or taking pains to avoid heap corruption is often considered not worth the cost.

For each of the listed mis-uses, try it out on your favorite compiler and describe how it appears to be handled. Then take the same misbehaving program and run it under Purify and see what Purify has to tell you about it. Does Purify catch each of these listed errors? How is each reported?

a) What happens when you try to `malloc` a region of 0 bytes?

b) What if you attempt to realloc a pointer that points into the stack?

c) What if you write off the end of a heap-allocated block?

d) What happens when you free a `NULL` pointer?

e) What if you try to free a string constant?

f) What if you free something in the stack?

g) What if you free a heap pointer twice?

h) What if you attempt to use a pointer after it has been freed? Do any of its contents immediately change upon being freed?

Note that in some cases, you might see no visible effects, which means you don't really know if something evil happened. At best, it means that `malloc`/`free` detected that your request was invalid and ignored it. At worst, it means you have subtly corrupted the heap and only later will the program happen upon the damage.

One special note about `free`. A behavior of the Sparcstations, common to many UNIX versions, is that when you free a pointer, nothing is done right away, and only on the next call to `malloc` is the previous free request actually committed. This allows a little

"grace period" after calling free in which your pointer remains valid. If you're testing on a UNIX machine, for the above questions which ask about the effects of free, you will want to put in a follow-on call to `malloc` in order to push through your earlier request completely before you can tell what the effects are.

Problem 10: Code Generation

Take a look at the functions below (by the way, this is all nonsense code, don't try this at home):

```
struct vegetable {
    int **roots;
    short leaves[2];
};

struct garden {
    float zucchini;
    struct vegetable onions;
    double *cucumber[2];
};

void main(void)
{
    struct garden *backyard;

    backyard->zucchini = Grow((char **)(backyard->cucumber[0]), &backyard[2]);
}
```

- Generate code for the entire `main` function.

```
int Grow(char **broccoli, struct garden *corner)
{
    short peppers[2];
    int *spinach;

    corner->onions.roots = &spinach;                                  // Line 1
    spinach[*spinach] = *((*(struct vegetable *)peppers).leaves); // Line 2
    return *(int *)(broccoli + peppers[0]);                           // Line 3
}
```

- Draw the activation record for `Grow` giving the size, type, and layout of the fields.
- Generate code for the `Grow` function. Mark the sequence of generated instructions with the line number of the C code to help comment your work. Assume the values of registers are not preserved between snippets. You do not have to try to optimize the generated code, a simple straightforward translation is all we ask.

Problem 11: Compilation process

The preprocessor

The following macro was designed to abstract away the icky pointer math to find the nth element of a DArray, adding to readability without the performance hit of a function call:

```
#define ADDRESS_OF_NTH(base, n, elemSize)  (char *)base + n*elemSize;
```

Here is how we might use such a macro in our `DArray` implementation:

```
void *ArrayNth(DArray darray, int index)
{
   assert(index >= 0 && index < darray->numElements);
   return ADDRESS_OF_NTH(darray->elements, index, darray->elemSize);
}
```

However, this first attempt at the macro is a bit sloppy and has several problems.

- Due to consequences of the way macros are expanded, this macro can compute the address incorrectly. Show a correct attempt to use the macro that will nonetheless result in an incorrect address because of a flaw in the macro itself. How would you change the macro to fix this problem?
- Look at the macro carefully and you'll find another careless error that can produce difficulties when compiling. Show a correct attempt to use the macro that will fail to compile because of another flaw in the macro. How would you change the macro to fix this problem?

You can see the effects of just running the preprocessor by giving the `-E` flag to `gcc` when compiling.

The compiler and linker

The items below describe several common compiling and linking errors. For each item, determine which tool reports the problem (the compiler or the linker or perhaps neither). If detected, describe the error message that is given and tell whether the error is a fatal error (stops the compilation process) or just a warning (you need to compile with the gcc flag `-Wall` to get all warnings, by the way). For the cases that go undetected or are just warnings, describe the effect that the code will have at runtime.

- Which tool detects if you make a call to a function without a visible declaration? (i.e. you call `printf` but haven't included the `stdio.h` header file)
- Which tool detects if you make a call to a function that has a visible declaration but no matching definition?
- Which tool detects if a single file uses the same name for two different functions? What if the functions with the same names are in two different files? Does it matter if the two functions have the same prototype?
- Which tool detects if you pass a function pointer of the wrong type (such as passing `strlen` to `ArrayMap`)? What happens if you add a correct typecast on the mismatched function pointer?

Problem 12: Drawing and understanding memory

Assume the following program is using a fully correct implementation of the DArray and it compiles without any warnings.

```
static void PrintString(void *elem, void *fp)
{
   fprintf((FILE *)fp, "%s\n", (char *)elem);
}

int main(void)
{
   char *names[4] = {"IBM", "Apple" ,"Microsoft", "H-P"};
   DArray companies;
   int i;

   companies = ArrayNew(sizeof(char *), 4, NULL);
   for (i = 0; i < 4; i++)
      ArrayInsertAt(companies, names[i], 0);
   ArrayMap(companies, PrintString, stdout);
   ArrayFree(companies);
   return 0;
}
```

Familiarize yourself with the code by tracing through it. Don't get caught up in whether you believe the code is right or wrong, just follow it as it stands. If the code cannot be traced because it crashes or has unpredictable results, indicate where you think it will have problems and why.

- Draw the state of the stack and heap right after the for loop and before the `ArrayMap` call. The drawing should show you understand what's on the stack, what's in the heap, what's a string constant, and what the storage for the `DArray` looks like at that time. Show the output that will be printed and explain how you arrived at your answer.