Matthew Lieberman
mrl196
185003809

Compiling/Running:
In order to run this program you must while in the folder in your terminal run the following command to compile the program:

```
make
```

if the program has been previously compiled you can run the command:

```
make clean
```

to clean all .o files from the folder

After the program has been compiled you can run the program by doing:

```
./memgrind
```

After running the program you will be presented with the option of displaying memory info, if the user chooses yes this can greatly impact run time

```
y or n
```

After running the program there will be an output giving the average of the time of each of the tasks being run 50 times, and below the tasks time averages there is a final check of memory to let the user see if there is any memory still being allocated

Functionality:

- Verify that mymalloc can allocate memory and myfree can free the allocated memory without causing any errors or memory leaks
- Test different memory allocation scenarios, such as allocating and freeing small and large memory blocks, allocating and freeing memory in a random pattern, etc., to ensure the library can handle arbitrary memory allocation and deallocation patterns
- Ensure that mymalloc correctly handles out-of-memory scenarios, such as when no free block of memory is available to allocate
- Verify that myfree handles invalid inputs, such as freeing a null pointer or a pointer not at the start of a block
- Verify that myfree can merge adjacent free blocks and minimize fragmentation of the memory pool

Testing:

the memgrind.c program has the following 5 testing tasks:

1. malloc() and immediately free() a 1-byte chunk, 120 times.
2. Use malloc() to get 120 1-byte chunks, storing the pointers in an array, then use free() to deallocate the chunks.
3. Randomly choose between
   • Allocating a 1-byte chunk and storing the pointer in an array
   • Deallocating one of the chunks in the array (if any)

4. Allocate and free a large chunk of memory
5. Allocate and free memory in a random pattern

If the user chooses to do so, each task can check /display the memory to let the user see at each step what the allocated memory looks like

Also if the user chooses to display information an additional 5 tests in err.c are done once:

1. attempts to free a memory location that was not allocated
2. frees a specific element of an allocated array, which is not valid
3. attempts to free the same memory block twice
4. attempts to free a null pointer
5. attempts to free a different pointer that points to the same memory block

Outputs:

Free null pointer:

```
printf("%s:%d: Can't free a NULL pointer\n", file, line);
```

Not enough free memory:

```
printf("%s:%d: Not enough memory\n", file, line);
```

Pointer not at start:

```
printf("%s:%d: Pointer is not at start of block\n", file, line);
```

Block already freed:

```
printf("%s:%d: Pointer is already freed\n", file, line);
```

Memory leak:

```
printf("Memory leak is detected: %d unfreed block(s)\n", num_leaks);
```

No Memory leak:

```
printf("No memory leaks are detected\n");
```

Print Memory:

```
printf("block at %p, size %lu, allocated: %s\n", (void *)current, current->size, current->allocated ? "yes" : "no");
```

Time of tasks:

You can see below that printing the memory can greatly increase the time of this program

Time with printing memory:	*Time without printing memory:*
Task 1: 201419 microseconds	Task 1: 187 microseconds
Task 2: 7813094 microseconds	Task 2: 19142 microseconds
Task 3: 386303 microseconds	Task 3: 588 microseconds
Task 4: 465159 microseconds	Task 4: 75504 microseconds
Task 5: 408585 microseconds	Task 5: 9133 microseconds