Allocation

Implement a dynamic heap allocation library, replacing the functionality of `malloc`/`free`, but with a different interface, as specified in this header:

chunk.h

```
#include <stddef.h>

void *chunk_create(size_t size);
size_t chunk_resize(void **p, size_t size);
size_t chunk_allocated(void *p);
size_t chunk_available(void *p);
void chunk_destroy(void *p);
```

A barebones `chunk.c` file will be provided to suggest a header format and helper functions; fill in the functions as suggested by comments. A test program that exercises the various library functions will be provided in `main.c`; build the whole thing into a program called `queue` with this Makefile:

Makefile

```
CFLAGS=-Wall -g

OBJECTS=\
  chunk.o \
  main.o

HEADERS=\
  chunk.h

chunk: $(OBJECTS)
chunk.o: chunk.c $(HEADERS)
main.o: main.c $(HEADERS)

.PHONY: clean
clean:
        rm -f chunk $(OBJECTS)
```

The output will depend on the size of a `size_t` on your system; on mine, the test produces the following output:

```
$ ./chunk
Allocating 30 bytes for a.
  a = 4 (32 bytes)
Allocating 20 bytes for b.
  b = 40 (20 bytes)
Destroying a.
Allocating 40 bytes for c.
  c = 64 (40 bytes)
Allocating 5 bytes for d.
```

```
  d = 4 (8 bytes, 32 available)
Allocating 5 bytes for e.
  e = 16 (8 bytes, 20 available)
  (d now has 8 bytes available)
Destroying e.
  (d now has 32 bytes available)
Destroying b.
  (d now has 56 bytes available)
Resizing d to 15 bytes.
  d = 4 (16 bytes, 56 available)
Resizing d to 60 bytes.
  d = 108 (60 bytes, 60 available)
Destroying c.
Destroying d.
Done.
```

Details

Each chunk, whether allocated or free, will begin with a fixed-size header of type `struct chunk`. The header encodes the size of the chunk (including the header) in terms of `struct chunk`s (not bytes), and also a bit flag indicating whether the chunk is in use (allocated) or not.

To allocate a new chunk in `chunk_create`, first compute the actual size needed (expressed in `struct chunk`s not bytes and including a header; a function is provided that does this for you). Then, starting at the beginning of the heap, find the first chunk that is both not in use and big enough, merging adjacent free chunks together before checking their size. This strategy is called 'first-fit'. Note that you need not even use a free list; this allocator will be correct and useful, but not highly optimized, and it is okay to do an $O(n)$ scan through the heap for each allocation.

If the chunk found is larger than needed, split it into a chunk just big enough, with the remainder forming a new second chunk. If no chunk is found that is big enough and unused, grow the heap to create such a chunk and use it. Finally, return a pointer to the memory just after the chunk header. In other functions, when passed a user pointer back in, find the chunk header with something like `(struct chunk *)p - 1`.

Resources

You will definitely want the manual page for the `sbrk` function, in order to find and grow the stack. A fascinating example of a real-world `malloc` implementation is dlmalloc, which has a lot of interesting comments in the code (but it is not useful for this assignment, because it is a different kind of allocator).

When you have finished, submit your `chunk.c` on CMS.