CS24 Midterm 2016 – COVER SHEET

This midterm requires you to write several programs, and to answer various questions. As with the homework assignments, a tarball is provided, **`cs24mid.tgz`**. When you start the midterm, you should download and extract this tarball and work in the directories specified by the problems.

When you have finished the midterm, you can re-archive this directory into a tarball and submit the resulting file:

> *(From the directory containing* ***`cs24mid`****; replace* ***`username`*** *with your username.)*
> **`tar -czvf cs24mid-`*`username`*`.tgz cs24mid`**

Important Note!

If you decide to not follow the filenames specified by the midterm problems, or your tarball is really annoying in some way (e.g. the permissions within the tarball have to be overridden by the TAs before they can grade your work!), you may lose some points. If you think you might need help creating this tarball properly, talk to Donnie or a TA; we will be happy to help.

Midterm Rules

The time limit for this midterm is 6 hours, in multiple sittings. The main rule is that if you are focusing on the midterm, the clock should be running. If you want to take the entire midterm in one sitting, or if you want to take a break (or a nap) after solving a problem, do whatever works best for you. Just make sure to track your time, and only spend 6 hours working on the exam.

If you need to go overtime, just indicate where time ran out. You receive half-credit for the first hour of overtime, no credit thereafter.

You may use any official course material on the exam. You may refer to the book, the lecture slides, your own assignments, solution sets, associated reference material, and so forth.

No collaboration with students, TAs, or others. You may not talk to another student about the contents of the midterm until both of you have completed it. You may talk to a TA about the midterm after you have completed it. You may also talk to a TA if you need help packaging up your midterm files, as mentioned above. Obviously, do not solicit help from others on the exam either.

Request any needed clarifications directly from Donnie. The TAs aren't responsible for writing the midterm. Feel free to contact Donnie if you have any questions. If you can't make any progress while you wait for an answer, stop your timer.

You may use a computer to write, compile, test, and debug your answers. In fact, this is encouraged, because we will deduct points for incorrect programs. Some problems include test code you can use to try your answers, although they may not detect all bugs (or any style issues!).

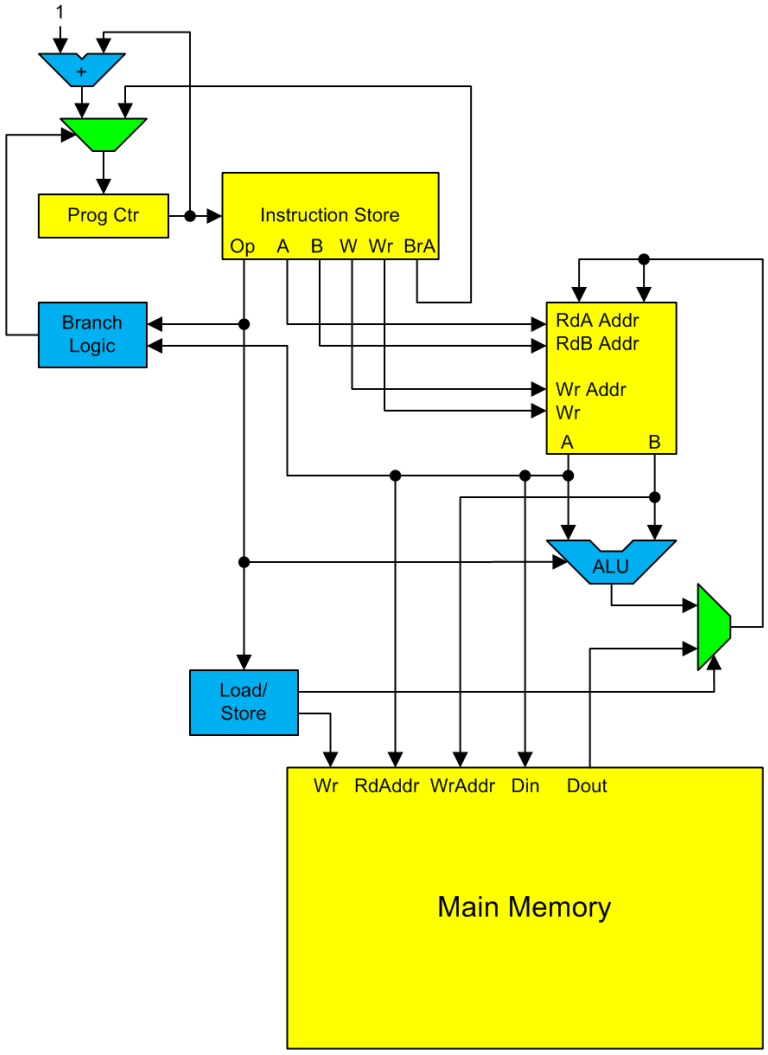
You may not use a disassembler, or the `gcc -S` feature, on the exam. The IA32 assembly code you turn in should be entirely self-written by you.

WHEN YOU ARE READY TO TAKE THE MIDTERM, YOU MAY GO ON TO THE NEXT PAGE! GOOD LUCK!

Problem 1: Bus Arbiter (24 points)

(The files for this problem will go in the **`arbiter`** subdirectory of the archive.)

Here is the simple processor we looked at in Lecture 2, which follows the Load/Store Architecture:



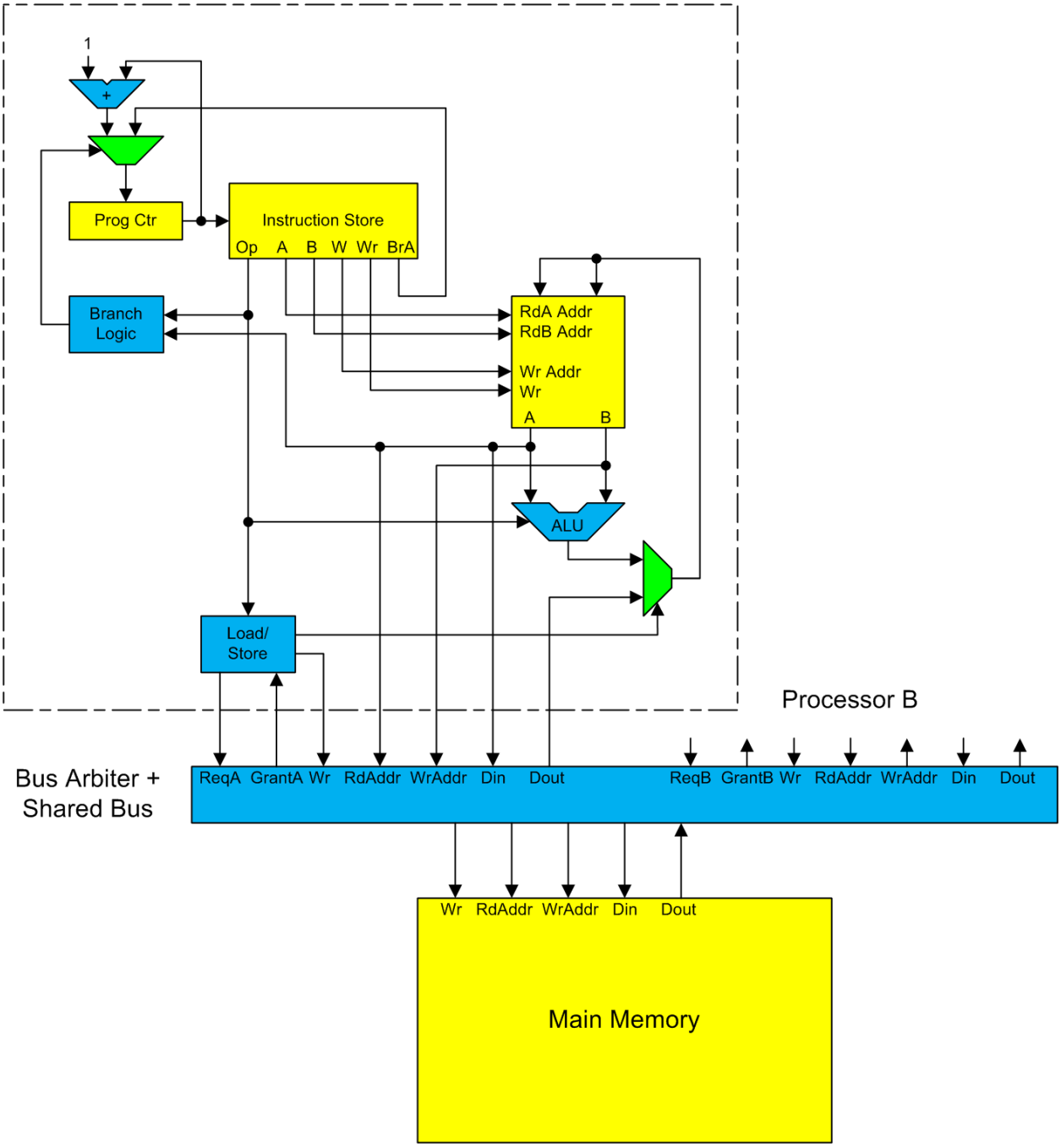
As discussed in this lecture, Load/Store processors do not feed values from main memory directly into the ALU; rather, there are separate **`LD`** (load) and **`ST`** (store) instructions for moving values between main memory and the register file.

In order to improve the performance of our computer, we would like to add a second processor to the system, allowing both processors to work with main memory at the same time. However, once we have two processors accessing the same memory, interactions must be coordinated to ensure the system doesn't end up in an invalid state.

This kind of problem is frequently solved through the use of a *bus arbiter*. Multiple devices (CPUs, in our case) are connected to a shared bus, along with an arbiter. When a CPU wants to use the bus, it makes a *request* to the arbiter. When the arbiter decides that the CPU can use the bus, the arbiter

will *grant* the request, telling the CPU that it has sole access to the bus. When the CPU is finished using the bus, it indicates this to the arbiter, and then another CPU can gain access to the bus.

An updated diagram would look something like this:



You can see that the Load/Store logic has been modified to interact with the bus arbiter. When a memory access is initiated, the Load/Store logic will send a signal to the arbiter. When the arbiter grants access to the processor, it will indicate this by sending a signal back to the Load/Store unit. It is important to note that while a processor is waiting to obtain access to the shared bus, it is

effectively "on hold," doing nothing until the arbiter allows it to continue. (It is unimportant how this capability is implemented, but most general-purpose processors have the ability to do this.)

Of course, the big challenge is when multiple devices want to use the bus at the same time. The arbiter must decide how to resolve multiple requests. There are many different ways that an arbiter can do this, some simple and unfair (i.e. some devices always receive access to the bus first), and others that are fairer yet more complex. We will implement a fair arbiter, which gives each processor a chance to access the bus in turn. The arbiter will store an internal memory value, which continually flips between Processor A and Processor B. (Since we have only two processors sharing the bus, we can do this with a single bit.) We will call this value "Turn" since it specifies which processor's turn it is to access the bus. When Turn = 0, Processor A has an opportunity to access the bus; when Turn = 1, Processor B has an opportunity to access the bus. Note that a processor is not *required* to access the bus when it's the processor's turn, but when a processor *wants* to access the bus, it must wait until its turn has arrived before it will be granted access by the arbiter.

As long as all Request inputs are 0, all Grant outputs should be 0, and the Turn value should simply toggle back and forth between 0 and 1. However, if a processor sets its Request signal to 1, then once it becomes that processor's turn, the Grant output to that processor should be set to 1 until the processor changes its Request signal back to 0. In addition, as long as a processor holds the bus, the Turn value should not change to the other processor. However, once the processor lets go of the bus, the Turn value flips back to the other processor and we continue on.

Here is a simple example where Processor B requests access to the bus:

Turn Value	ReqA	ReqB	GrantA	GrantB	Details
0 (Proc A)	0	0	0	0	No processors requesting access, so the arbiter simply switches the Turn value back and forth.
1 (Proc B)	0	0	0	0	
0	0	0	0	0	
1	0	0	0	0	
0	0	1	0	0	Processor B requests the bus, but it's not that processor's turn yet.
1	0	1	0	1	It becomes Processor B's turn, so the arbiter grants the request.
1	0	1	0	1	Time passes while Processor B does its thing. Note that the Turn value stops toggling while a processor has the bus.
1	0	0	0	1	Processor B releases the bus. Arbiter will resume toggling.
0	0	0	0	0	Arbiter goes back to toggling between processors.
1	0	0	0	0	
0	0	0	0	0	
…	…	…	…	…	

Here is a more complex example where Processor B requests access to the bus, and then Processor A also requests access:

Turn Value	ReqA	ReqB	GrantA	GrantB	Details
0 (Proc A)	0	0	0	0	No processors requesting access, so the arbiter simply switches the Turn value back and forth.
1 (Proc B)	0	0	0	0	
0	0	0	0	0	
1	0	0	0	0	
0	0	1	0	0	Processor B requests the bus, but it's not that processor's turn yet.
1	0	1	0	1	It becomes Processor B's turn, so the arbiter grants

					the request.
1	0	1	0	1	Time passes while Processor B does its thing. Note that the Turn value stops toggling while a processor has the bus.
1	1	1	0	1	Processor A now requests the bus. However, since it's not Processor A's turn, the arbiter makes Processor A wait.
1	1	0	0	1	Processor B finally releases the bus. Arbiter will resume toggling.
0	1	0	0	0	Arbiter toggles back to processor A's turn…
0	1	0	1	0	Now the arbiter grants Processor A access to the bus.
0	1	0	1	0	As before, as long as Processor A holds the bus, the arbiter keeps the Turn value on Processor A.
0	0	0	1	0	Finally, Processor A releases the bus.
1	0	0	0	0	Arbiter goes back to toggling between processors.
0	0	0	0	0	
…	…	…	…	…	

Initially, we will simply add the request/grant logic to the **`LD`** (load) and **`ST`** (store) instructions. When the processor wants to perform a load or store operation, the processor will wait for the arbiter to grant access to the bus.

(a) You will find a simple implementation of a bus arbiter in **`arbiter.c`**. Find the **`clock_arbiter()`** function and fill in the logic that updates the GrantA, GrantB, and "Next Turn" values, based on the Current Turn value and ReqA/ReqB inputs. Make sure to clearly document your work. *(8 points)*

The program **`testarb.c`** will exercise the arbiter by feeding it various inputs. It will print error messages if both processors are granted the bus at the same time, but other than that, you must manually check the behavior of the arbiter. You can build the test program with the provided makefile, and then run it from the command line: **`./testarb`**

The arbiter test code uses a random number generator to drive the input signals, but the seed is always the same, so the test run will always be the same. If you want to try different test runs, you can specify a different random number seed on the command-line, such as "**`./testarb 12345`**"

(b) This design is very simple because **`LD`** (load) and **`ST`** (store) will automatically acquire access to the shared bus before proceeding. However, we see an undesirable performance issue, because now at least one extra clock-cycle is imposed for every single **`LD`** or **`ST`** operation we perform, even if there is no bus contention, because every single **`LD`** or **`ST`** requires the arbiter to grant the bus-request, and the arbiter will take one or more cycles to respond. Even two sequential **`LD`**/**`ST`** operations require independent interactions with the bus-arbiter.

Assume that all instructions would normally take only one clock cycle to complete. (This includes **`LD`** and **`ST`** *before* we introduced the bus arbiter; with the arbiter, now they will take two or more cycles.) Here is a simple assembly program to sum two vectors into a third output-vector, along with the input registers:

R0 = start of array A
R1 = start of array B

R2 = length of arrays
R7 = start of array Result

```
    BZ  R2, EndLoop  # if (R2 == 0) goto EndLoop;
Loop:
    LD  R0, R3       # R3 = Memory[R0]
    LD  R1, R4       # R4 = Memory[R1]
    ADD R3, R4, R5   # R5 = R3 + R4
    ST  R5, R7       # Memory[R7] = R5
    INC R0           # Move ptr to next element of A
    INC R1           # (Same with B, and with Result)
    INC R7
    DEC R2           # Decrement length...
    BNZ R2, Loop     # if (R2 != 0) goto Loop;
EndLoop:
    DONE
```

Analyze the impact of introducing our bus arbiter on the performance of this program. In a file **`part_b.txt`**, answer these two questions. *(8 points)*

- If the array length is 100, how much longer will the program take to run on the multi-processor computer with the bus arbiter, as compared to the original processor? Assume that only one CPU of the multi-processor is interacting with the main memory.
- If both CPUs in the multi-processor are executing this same program, how much worse could the performance become, in the worst case? Make sure to state any assumptions in your analysis.

(c) In order to mitigate the performance penalty of having every single load/store operation acquire the bus, the processor designers want to introduce a new pair of instructions: **`BLK`** will acquire access to the shared bus ("Bus-LocK"), and **`BUN`** will release access to the shared bus ("Bus-UNlock"). Now, instead of each **`LD`** or **`ST`** instruction acquiring access to the bus, we can enclose a series of **`LD`** and **`ST`** operations with **`BLK`** and **`BUN`**, to allow the processor to perform multiple load and store operations without having to ask the arbiter for access on every single operation. **`LD`** and **`ST`** go back to taking one clock cycle to complete, and **`BLK`** and **`BUN`** now take two or more cycles to complete.

In a file **`part_c.txt`**, take the program from part (b) and suggest how it should be modified to use the **`BLK`** and **`BUN`** instructions in the program to improve its performance from what we saw in part (b). Justify your strategy by briefly analyzing how the program's performance would improve, both in the case where a single CPU is running the program, and also when both CPUs are running the program. *(8 points)*

Problem 2: Linked List Reversal (36 points)

In the directory **`cs24mid/reverse`**, you will find a simple program that builds a linked list of integers, and then attempts to reverse the list. This doesn't work yet, because the list-reversal operation hasn't yet been implemented.

You must use IA32 assembly language to implement a function **`reverse_list(LinkedList *)`** that performs an *in-place* reversal of the list. That is, you may not generate a new list that holds the

original list's values; you must change the various pointers in the list to reverse its contents in-place.

This turns out to be reasonably straightforward to do, except for the fact that you are doing it in assembly code.

Your Tasks

1. *(8 points)* In the file **`reverse/design.txt`**, write high-level pseudocode for the reverse-list operation. Don't forget that the head- and tail-pointers of the list need to be reversed, as well as the list elements themselves.

   If you have difficulty figuring out where to start, draw a picture of a linked list with a few elements in it. Then, in another color, update the arrows between the various parts of the list to reverse its contents. This should make it very easy to see what you need to do. (Please don't submit the picture.)

2. *(8 points)* If your pseudocode contains various higher-level flow-control constructs like if-statements or do/while/for loops, duplicate it after your answer to part 1 (don't overwrite part 1), then translate the higher-level flow-control constructs into simpler operations that can be performed in IA32 assembly language, like gotos and simple tests. At this point, your pseudocode should be relatively straightforward to translate into IA32 assembly.

3. *(20 points)* In the file **`reverse.s`**, implement your **`reverse_list(LinkedList *list)`** function in IA32 assembly language. Here are some additional guidelines:

   - You can assume that your function will always receive a non-**`NULL`** pointer.
   - The list passed to the function may be empty, (i.e. **`list->head == NULL && list->tail == NULL`**); your function should handle this situation properly as well.
   - You will need to add **`reverse.o`** to the **`Makefile`** in order to get it to build.

   You should also make sure to follow good coding style and formatting practices. Make sure to comment your code well (don't just repeat what the code says; explain what is going on), and include a comment header at the top of your function that explains what the function signature is, what the function does, where the arguments are on the stack, etc.

   Once you have completed your implementation, you can test it with the provided files. Run the tester multiple times, since it is feasible (albeit highly unlikely) that a palindromic list could be randomly generated. If the tester reports no errors, congratulations!

Problem 3: Small Objects (40 points)

*In a file **`cs24mid/smallobj.txt`**, answer the questions in the below discussion.*

Consider a program that heap-allocates and releases *many* small blocks of memory (a.k.a. "small objects"), all blocks being approximately the same size. The heap is managed by an explicit allocator, so the program must also release these blocks, but let's assume the program does this correctly. For the sake of concreteness, let's assume that no block is larger than 100 bytes, and as stated before, all allocations are approximately the same size.

These small objects are used for a certain period of time and then released, so the overall memory usage of the program stays approximately constant, but the program is continuously allocating more new objects to use, and also freeing objects it is finished with. At any given point, there might be tens of thousands to hundreds of thousands of these small objects in memory.

1. *(6 points)* In class we discussed simple general-purpose allocators based on either implicit or explicit free-lists. If the allocator uses a free-list to manage the heap, will the described behavior cause allocation failures due to memory fragmentation? Why or why not?

2. *(6 points)* Enumerate other issues or inefficiencies that would be produced by this combination of allocator and program behavior.

The program is modified so that instead of directly allocating these small objects from the heap, the program uses a "small object allocator" that is built on top of the standard allocator. The small-object allocator uses a number of large "chunks" that are themselves allocated from the heap. Each chunk can hold *N* small objects (where *N* is large, say thousands), so the internal structure of the chunk becomes very simple:

```
struct chunk {
    /* The size of small objects in this chunk. */
    unsigned int object_size;

    /* The total number of objects the chunk can hold. */
    unsigned int total_objects;

    /* Index of the next available object in the chunk. */
    unsigned int next_available;

    /* How many small objects in this chunk have been released. */
    unsigned int num_freed;

    /* The start of the memory area for this chunk. */
    char mem[];
};
```

These chunks are then managed by the small-object allocator, with the idea that there will be a small number of large chunks that can hold a large number of small objects.

We won't go into all the details of how chunks are managed by the allocator, but the general principle is that it will create a new chunk when more space is needed for small objects, and that chunk will service a fixed number of allocation requests (i.e. the value in **`total_objects`**). When the chunk has been completely filled up, then the small-object allocator must create another chunk to service allocation requests.

When a small object is freed, the small-object allocator doesn't actually reclaim the memory for that object; instead, it simply records that one more object has been freed in the chunk. In other words, the chunk's **`num_freed`** value is incremented.

When a chunk has no more room for allocations, and all of the chunk's objects have been freed, it can be reset and reused by the allocator for more allocation requests. If a large number of empty chunks build up, some chunks can be released back to the memory heap.

Let's look at a few more details of how chunks can be used. A chunk can be initialized as follows:

```
chunk * init_chunk(unsigned int total_objects, unsigned int object_size) {
```

```
    chunk *c = malloc(sizeof(chunk) + total_objects * object_size);
    c->object_size = object_size;
    c->total_objects = total_objects;
    c->next_available = 0;
    c->num_freed = 0;
}
```

Allocation and deallocation against chunks are performed as follows:

```
/* Attempt to allocate another small object from this chunk.  If
 * there is no more room in the chunk, NULL is returned; otherwise
 * a pointer to a memory block is returned for the caller to use.
 */
void * smallobj_alloc(chunk *c) {
    void *obj = NULL;
    if (c->next_available < c->total_objects) {
        /* Compute a pointer to the next available memory region. */
        obj = (void *) (c->mem + c->obj_size * c->next_available);

        c->next_available++;
    }
    return obj;
}

/* We don't actually keep track of freed memory regions with the
 * intent to use them again; we just record that an object was freed.
 * We don't reclaim any of the chunk's memory until all small objects
 * are no longer available.
 */
void smallobj_release(chunk *c, void *obj) {
    /* Make sure the small object is actually from this chunk! */
    assert(is_from_chunk(obj, c));

    /* Make sure the pointer points to the start of an object. */
    assert((int) (obj - mem) % c->obj_size == 0);

    c->num_freed++;
}

/* A helper function to tell us if a particular object came from the
 * specified chunk.
 */
int is_from_chunk(void *obj, chunk *c) {
    return (obj >= c->mem && obj < c->mem + c->total_objects);
}
```

As you can see, this is a very simple allocator, so simple that it doesn't make any attempts to reuse memory that a small object previously occupied, until *all* objects in the chunk have been released!

3. *(10 points)* Given the program's allocation patterns, what benefits and/or issues will arise from using this "small object allocator," as compared to using the general-purpose heap allocator for small objects? Be complete in your answer.

The program's original allocation behavior was that even though many tens or hundreds of thousands of objects are allocated at any given time, objects are only kept in memory for a relatively short time and then are released. Let's say that certain changes are made to the program, and now its behavior is slightly different: now, *most* of the objects still follow the same pattern, but 5-10% of the objects have a much longer lifetime before being deallocated.

4. *(6 points)* How will the small-object allocator described above fare with this kind of allocation behavior? What issues or problems are likely to emerge?

5. *(12 points)* Describe how the small-object allocator should be modified to mitigate any issues that you identify.

    - You don't have to write out a bunch of code, but your description should be at the level that a suitably capable programmer could take it and know exactly how to update the small-memory allocator to resolve issues you identify in the previous question.
    - If changes are necessary to the data structures, you should specify what these changes are.
    - You should also explain how your proposed changes will help to mitigate the issues identified in the previous problem.