CS3214 Fall 2024 Final Exam Solutions

December 15, 2024

Contents

1 Networking (32 pts) 2
- 1.1 Know Your Internet (10 pts) . . . 2
- 1.2 HTTP (22 pts) . . . 3
  - 1.2.1 Chunked Transfer Encoding (5 pts) . . . 3
  - 1.2.2 Fill In The Blanks (11 pts) . . . 3
  - 1.2.3 HTTP vs HTTPS (6 pts) . . . 4

2 Virtual Memory (20 pts) 5
- 2.1 `malloc` vs. `calloc` (16 pts) . . . 5
- 2.2 Demand Paging (4 pts) . . . 6

3 Dynamic Memory Management (22 pts) 7
- 3.1 Comparing Allocators G and T (8 pts) . . . 7
- 3.2 Simple Segregated Storage (5 pts) . . . 8
- 3.3 V-Table Exploits (9 pts) . . . 8

4 Automatic Memory Management (18 pts) 9
- 4.1 Reachability (14 pts) . . . 9
- 4.2 GC (4 pts) . . . 11

5 Virtualization (8 pts) 12

1 Networking (32 pts)

1.1 Know Your Internet (10 pts)

Determine whether the following statements related to networking are true or false.

(a) The Internet is an interconnected network of networks.

☑ true / ☐ false.

(b) In packet-switched networks where data is sent in chunks called packets, a node can simultaneously transmit multiple packets on a single outgoing point-to-point link using multiplexing.

☐ true / ☑ false.

In a packet-switched network, packets use the full bandwidth of the link while they're being transmitted.

(c) "Best effort" delivery refers to a network service model in which packets may be lost, duplicated, or arrive out of order.

☑ true / ☐ false.

(d) Internet core routers perform forwarding when a packet reaches their destination but the intended recipient has moved to a different node.

☐ true / ☑ false.

Forwarding refers to the act of choosing an outgoing link based on the routing table and then sending the packet on that link.

(e) When autonomous systems (AS) operated by different organizations interconnect, administrative policies may decide whether or not to send packets along advertised routes.

☑ true / ☐ false.

(f) In order to achieve maximum utilization, TCP connections must increase their window size as the propagation delay between the endpoints increases.

☑ true / ☐ false.

(g) The networking community requires that when new transport protocols are proposed that provisions to protect the network from congestion are included.

☑ true / ☐ false.

(h) Application layer protocols that use TCP place their application-level protocol messages inside TCP segments, which are delivered to the receiving peer as one entity.

☐ true / ☑ false.

TCP does not preserve any message boundaries. See RFC 9293 Section 10

(i) In wired networks, the primary cause of packet loss is due to routers dropping packets when their receive queues overflow under heavy traffic.

☑ true / ☐ false.

(j) TCP uses sequence numbers that start at zero and are increased by the number of words sent in a TCP segment.

☐ true / ☑ false.

TCP sequences number don't start at zero, but at a randomly chosen number, and they are counted in bytes, not words.

1.2 HTTP (22 pts)

1.2.1 Chunked Transfer Encoding (5 pts)

RFC 9112 describes a so-called “Chunked Transfer Encoding” where the body of an HTTP response consists of multiple chunks of varying lengths. This encoding is announced using a “Transfer-Encoding” header. Each chunk is preceded by its length l in bytes on a separate line, then followed by the chunk's content (l bytes), then followed by an extra CRLF (carriage return + line feed). When all data has been transmitted, a chunk of size 0 is sent.

Here is an example of such a response where the CRLF bytes (in the body) were made visible using their customary C-style escapes `\r\n`.

```
HTTP/1.1 200 OK
Content-Type: text/plain
Transfer-Encoding: chunked

6\r\n
CS3214\r\n
16\r\n
Computer Systems\r\n
0\r\n
\r\n
```

(a) (1 pts) Is this transfer encoding method 8-bit clean (that is, allows for arbitrary bytes/octets to be sent, such as UTF-8 encoded Unicode?). Say why or why not.

> Yes. Since each chunk has a length, the receiver can count out exactly as many bytes, without needing any kind of character escaping, hence all possible byte values can be used.

(b) (1 pts) When a chunked transfer encoding is used, which otherwise necessary header now becomes obsolete?

> The `Content-Length` header is now obsolete.

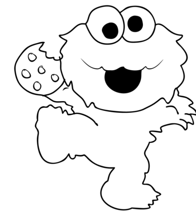
(c) (2 pts) What is a key benefit of this transfer encoding when compared to the standard transfer encoding used in p4?

> The key benefit is that the sender can start sending without knowing how long the entire response will be (because it's not required to compute the length of the entire response in order to send the content-length header).

(d) (1 pts) Can this transfer encoding be used with persistent connections in HTTP/1.1. Say why or why not.

> Yes. The final chunk (with zero length) concludes the response, and the client knows exactly when the next response in the stream will start.

1.2.2 Fill In The Blanks (11 pts)

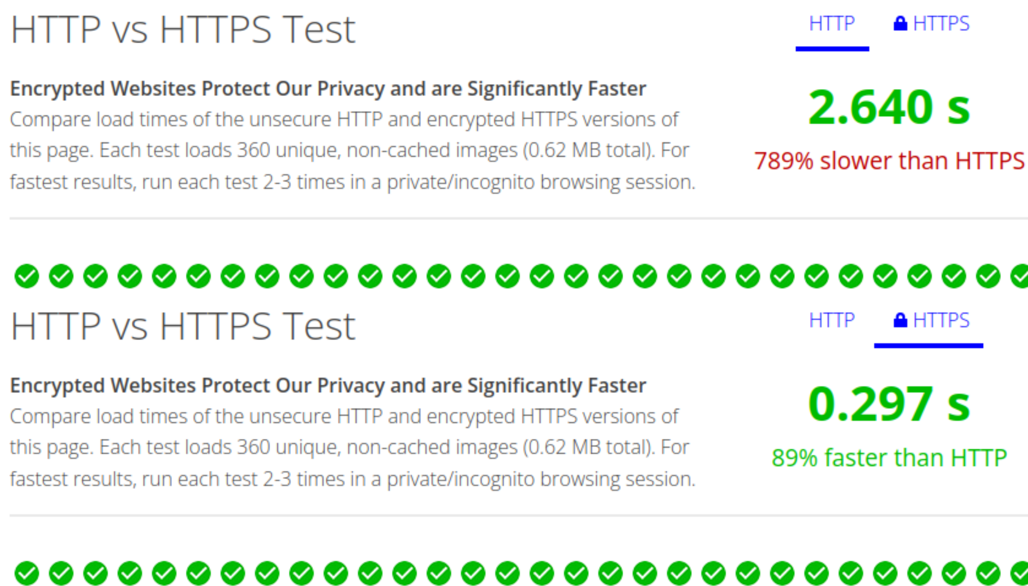


The design of the HTTP protocol is based on individual transactions, each of which involves a request and a response. Multiple of these transactions may occur via the same transport layer connection. Servers, however, are not required to remember previous transactions on an existing connection, which makes HTTP a stateless protocol.

When an HTTP server needs to recognize clients it previously served, it enlists the client's help. To sweeten the deal, it offers the client a cookie. When properly offered, the client accepts the server's offer and stores what it received in a storage area that is sometimes called a cookiejar (or cookie store). Later, when the client makes a new request to the same server, it will send a copy of what it previously received to the server. Servers must validate what they receive from clients in a way that is cryptographically secure (or sound). They must also ensure that clients cannot forge this information and pretend that they successfully engaged in prior transactions when in fact they did not. Clients typically will not hold onto this information forever, but instead let it expire after some time, at which point a human user attempting to use the website the server hosts may find themselves logged out and will need to provide their password (or credentials) again.

1.2.3 HTTP vs HTTPS (6 pts)

To counter the common perception that the use of the secure https protocol is slower than the use of the insecure http protocol, and to advocate for the use of encrypted websites, the website `www.httpvshttps.com` provides a demonstration for its visitors that runs a test where resources are loaded with and without https. 2 (partial) screenshots are shown below:



Explain what causes the performance difference between the two protocols. Note that this website assumes the use of a modern browser that supports the most recent versions of the HTTP protocol.

(a) (3 pts) http is slower in this experiment because ....

HTTP/1.1 uses only a limited number of connections and does not use pipelining on these connections (recall that although HTTP/1.1 theoretically allowed for pipelining, it is not generally used). As a result, the client can request at most one object/image per round-trip time, even with a very fast link.

(b) (3 pts) https is faster in this experiment because ....

The use of https, in a recent browser, enabled the use of HTTP/2, which means that all requests can be pipelined without needing to wait for an answer to previous requests. So the website is not quite honest in comparing http vs https, because the underlying performance improvements are really due to HTTP/2 — but as we discussed HTTP/2 is offered only in connection with a secure transport layer as per the policies of all recent browsers.

2 Virtual Memory (20 pts)

2.1 malloc vs. calloc (16 pts)

Consider the following C program where headers were elided for brevity:

```
1  int main()
2  {
3      size_t usize = 1<<21;
4
5      char cmd[128];
6      snprintf(cmd, sizeof cmd, "ps -o rss,vsz -p %d", getpid());
7      system(cmd);
8      uint8_t *arr = malloc(usize);
9      system(cmd);
10     memset(arr, 0xff, usize);
11     system(cmd);
12     free(arr);
13     system(cmd);
14 }
```

The **system(3)** function runs a Unix command (**ps** in this case), which outputs two pieces of virtual memory related information about the current process: (a) RSS, which represents that amount of physical memory a process is currently using, and (b) VSZ, which represents the amount of virtual memory addresses in use, both given in KB (kilobytes).

(a) (12 pts) When running this program, we obtain the output shown below in the left column. Complete this explanation chart below, adding explanations in the provide space for any changes (or lack thereof) for **both RSS and VSZ** on lines 9, 11, and 13:

```
(on line 7)      Explanation
  RSS    VSZ
 1024   2504     Baseline due to libraries, dynamic loader, etc.

(on line 9 it outputs:) (add explanation here)
  RSS    VSZ    | The change in VSZ corresponds to a request to allocate
 1024   4688    | 2MB of virtual memory or addresses.
                | There is no change in RSS since no physical memory
                | is allocated since on-demand paging is used
                |_____________________________________________________________
```

```
(on line 11 it outputs:) (add explanation here)
  RSS    VSZ    | The memset call will touch the allocated region, causing
 3160   4688    | the OS to fault in those pages and allocate about 2MB of
                | physical memory.
                |__________________________________________________________
```

```
(on line 13 it outputs:) (add explanation here)
  RSS    VSZ    | The program returns the virtual addresses as part of the
 1104   2636    | free() call, which calls munmap() under the hood.
                | The OS can also free any physical pages allocated.
                |__________________________________________________________
```

(b) `4 pts` We now change line 8 to use `calloc()` instead of `malloc()` like so:

```
8       uint8_t *arr = calloc(usize, 1);
```

Perhaps surprisingly, the output of the program doesn't change. Given what you know about virtual memory techniques, provide a possible explanation for why both programs exhibit the same memory consumption. **Focus only on the RSS/VSZ numbers reported on line 9.**

If implemented naively, `calloc` would obtain virtual memory from the OS and then touch it to zero it out. However, `calloc` is optimized and knows that the memory it receives from the OS already contains zeros when read, so it will not touch it to write zeros there. Moreover, the OS dedicates a single physical page frame that's filled with all zeros for the purpose of providing zero-initialized memory to applications like in this case. This page can be shared among all processes, so it's not counted in the per-process RSS. It uses a COW (copy-on-write) mapping, therefore actual physical memory consumption will start only once new values are written to those pages.

2.2 Demand Paging (4 pts)

In a discussion about demand paging (or on-demand paging, or lazy loading), a student on the CS3214 Discourse forum wrote:

> *The way demand paging improves virtual memory efficiency is through loading pages only when they are needed. This allows for more programs to run simultaneously with less physical memory. This also reduces initial memory usage and startup times.*

Discuss an example scenario where the use of on-demand paging will increase (and not reduce) program startup times, and why. Then explain if it is possible to avoid such scenarios and if so, how.

On-demand paging will result in slower startup times (when compared to loading all text pages into memory explicitly) if it results in many page faults, whose total cost in terms of processing and I/O time is larger than if the entire program text would have been read from secondary storage ahead of time. This is typically the case if the page faults are of the major variety (the data is not already cached and requires going to disk), and if the program accesses most of its text segment (rather than only a small subset).

3 Dynamic Memory Management (22 pts)

3.1 Comparing Allocators G and T (8 pts)

Consider the following C program.

```
1  static void allocate_pair(const char *label, size_t size)
2  {
3      void *block_1 = malloc(size);
4      void *block_2 = malloc(size);
5      printf("-- %s: %lu bytes --\n", label, size);
6      printf("Block 1: %p\n", block_1);
7      printf("Block 2: %p\n", block_2);
8      printf("Difference: %d\n\n", abs(block_2 - block_1));
9      free(block_1);
10     free(block_2);
11 }
12
13 int main() {
14     allocate_pair("A", 8);
15     allocate_pair("B", 128);
16     allocate_pair("C", 129);
17     allocate_pair("D", 160);
18 }
```

We will use 2 allocators, called G and T, respectively, to run this program.
When run with allocator G and allocator T, the program produced the following output:

Result using **Allocator G**:

```
-- A: 8 bytes --
Block 1: 0x55b9d45822a0
Block 2: 0x55b9d45822c0
Difference: 32

-- B: 128 bytes --
Block 1: 0x55b9d45826f0
Block 2: 0x55b9d4582780
Difference: 144

-- C: 129 bytes --
Block 1: 0x55b9d4582780
Block 2: 0x55b9d45826f0
Difference: 144

-- D: 160 bytes --
Block 1: 0x55b9d4582810
Block 2: 0x55b9d45828c0
Difference: 176
```

Result using **Allocator T**:

```
-- A: 8 bytes --
Block 1: 0x5595ad256008
Block 2: 0x5595ad256010
Difference: 8

-- B: 128 bytes --
Block 1: 0x5595ad262000
Block 2: 0x5595ad262080
Difference: 128

-- C: 129 bytes --
Block 1: 0x5595ad264000
Block 2: 0x5595ad264090
Difference: 144

-- D: 160 bytes --
Block 1: 0x5595ad266000
Block 2: 0x5595ad2660a0
Difference: 160
```

Compare and contrast the policies used by Allocator G and Allocator T based on the following aspects:

a) (2 pts) What is the amount of internal fragmentation for the allocation of A/Block 1?

`Allocator G`: 24 bytes

`Allocator T`: 0 bytes

b) (2 pts) What can you conclude about the use of Knuth's boundary tag technique?

`Allocator G`: ☑ Possibly / ☐ No / ☐ Cannot determine

`Allocator T`: ☐ Possibly / ☑ No / ☐ Cannot determine

c) (2 pts) What can you conclude about which insertion policy for free lists is used?

`Allocator G`: ☐ FIFO / ☑ LIFO / ☐ Cannot determine

`Allocator T`: ☐ FIFO / ☐ LIFO / ☑ Cannot determine

d) (2 pts) Does this allocator reuse free blocks immediately?

`Allocator G`: ☑ Yes / ☐ No / ☐ Cannot determine

`Allocator T`: ☐ Yes / ☐ No / ☑ Cannot determine

3.2 Simple Segregated Storage (5 pts)

The textbook by Bryant and O'Hallaron describes "simple segregated storage" as follows:

> With simple segregated storage, the free list for each size class contains same-size blocks, each the size of the largests element of the size class. (...) To allocate a block of some given size, we check the appropriate free list. If the list is not empty, we simply allocate the first block in its entirety. Free blocks are never split to satisfy allocation requests. If the list is empty, the allocator requests a fixed-size chunk of additional memory from the operating system, divides the chunk into equal-size blocks, and links the blocks together to form the new free list.

In no more than one paragraph, discuss the suitability of this storage scheme for embedded (and often memory-constrained) devices that may lack virtual memory support. If necessary, propose alternatives that may be better suited. Justify your arguments.

> This scheme would be ill-suited for memory-constrained environments because of the large potential for both internal and external fragmentation. Internal fragmentation can result from the fact that blocks are never split. External fragmentation results from the fact that chunks can be returned to the OS (and reused for other blocks) only if all blocks in that chunk have been freed. Segregated fits (like what most of you implemented in p3) provide a much better alternative here because they support splitting blocks and coalescing across size classes which can result in lower internal and external fragmentation.

3.3 V-Table Exploits (9 pts)

In object-oriented languages such as C++ or Java, objects that use virtual methods contain a header that points to a dispatch table for its virtual functions, commonly called a V-Table. In C++, which uses explicit memory allocation, this design is often the target of attacks that seek to exploit memory-related vulnerabilities, as demonstrated in the following C++ program:

```
1  #include <iostream>
2  #include <cstring>
3
4  struct GoodClass {
5    virtual void f() { std::cerr << "Nothing nefarious here." << std::endl; }
6  };
7
8  struct AttackersClass {
```

```
 9    virtual void f() { std::cerr << "You have been pwned." << std::endl; }
10  };
11
12  int main() {
13      GoodClass *gc = new GoodClass;  // allocate object of type GoodClass
14      gc->f();
15      delete gc;              // free object pointed to by gc
16      new AttackersClass;     // allocate object of type AttackersClass
17      gc->f();        // use-after-free error
18  }
```

when compiled without optimizations and run, the output is:

```
Nothing nefarious here.
You have been pwned.
```

We assume here that the execution of the object construction via `new` on line 16 was the result of an attacker's ability to divert control flow.

(a) (6 pts) The root cause of the vulnerability is of course the use-after-free error on line 17. Complete the output a user would see if they ran this program under valgrind:

```
==2930785== Invalid read of size 8
==2930785==    at 0x4011F7: main (wipevtable.cc:17)
==2930785==  Address 0x4da9c80 is 0 bytes inside a block of size 8 free'd
==2930785==    at 0x4848BE1: operator delete(void*, unsigned long) (vg_replace_malloc.c:1181)
==2930785==    by 0x4011E0: main (wipevtable.cc:15)
==2930785==  Block was alloc'd at
==2930785==    at 0x4844FB5: operator new(unsigned long) (vg_replace_malloc.c:487)
==2930785==    by 0x4011A8: main (wipevtable.cc:13)
```

(b) (3 pts) Describe how you would design a "hardened" explicit memory allocator that could defend against this type of attack. You may assume that the allocator is integrated with the C++ compiler when issuing calls to implement the `new` operator.

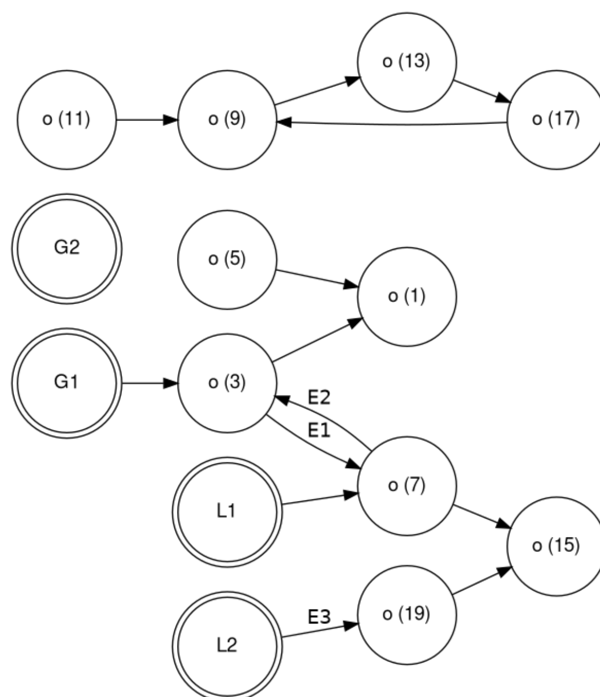
> Ensure that the allocator does not allocate objects of different types in the same space, ever. This way, the `AttackerClass` instance would not have reused the space where the `GoodClass` instance was allocated. This is called *Type Isolation* and is for instance used inside Apple's XNU kernel.[1] Another alternative is for the allocator to delay the reuse of memory in general, which would also have prevented this type of attack.

4 Automatic Memory Management (18 pts)

4.1 Reachability (14 pts)

The following diagram shows an abstracted depiction of a reachability graph. Nodes with a single circle are heap-allocated objects, and their size is given in parentheses. For instance, "o (5)" is an object of size 5. Nodes with a double circle are roots, which include global and local variables, e.g., "G2" is a global variable and "L1" is a local variable.

[1] See https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety/



Answer the following questions:

(a) (2 pts) Not counting the space taken up by the roots themselves, what is the total allocated memory at this instant? $\sum_{i=1}^{10} 2i - 1 = 10^2 = 100$

(b) (2 pts) If a full garbage collection took place, how much memory could be reclaimed?

$11 + 9 + 13 + 17 + 5 = 55$

(c) (2 pts) If a full garbage collection took place, how much memory would remain allocated after the collection has completed?

$100 - 55 = 45$

(d) (2 pts) What is the size of the subheap retained by L2? (The retained size is the amount of memory that is dominated by L2; that is, the amount that could be reclaimed were L2 set to null.)

19 (since "o (15)" is also kept alive by G1 and/or L1)

(e) (2 pts) What is the retained heapsize of G1?

0. (since none of the objects reachable from G1 are only reachable from it)

(f) (2 pts) Consider the edges labeled E1 and E2. Give a possible example of two Java statements that could have created edges E1 and E2.

For instance: `a.f = b; b.g = a;` (where a and b must be references to heap objects. Also possible is `a.f = b; b.f = a;`)

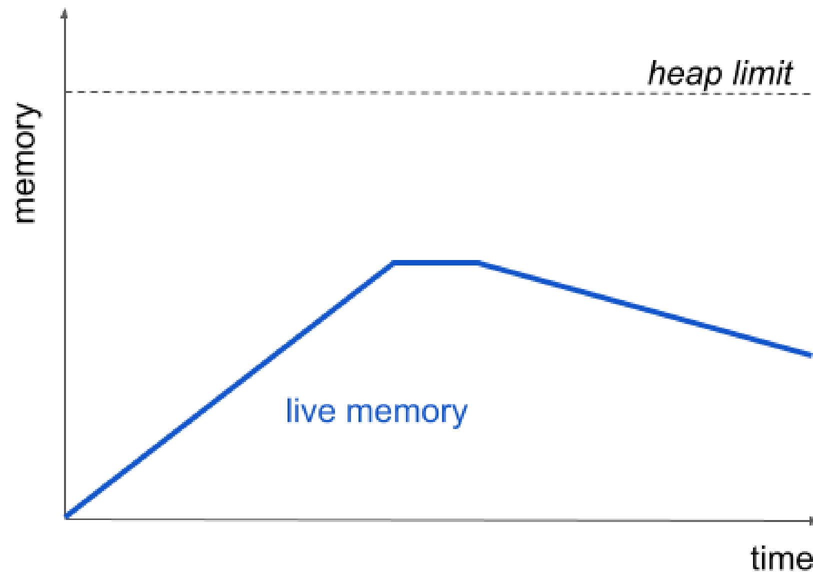
(g) (2 pts) Consider the edge labeled E3 (and remember that L2 is a local variable). Give a possible example of a Java statement that could have created edge E3.

This could be any statement like `var L2 = new O();` or `var L2 = r` where r is a reference, or without the `var` if it is noted that `L2` denotes a local variable.

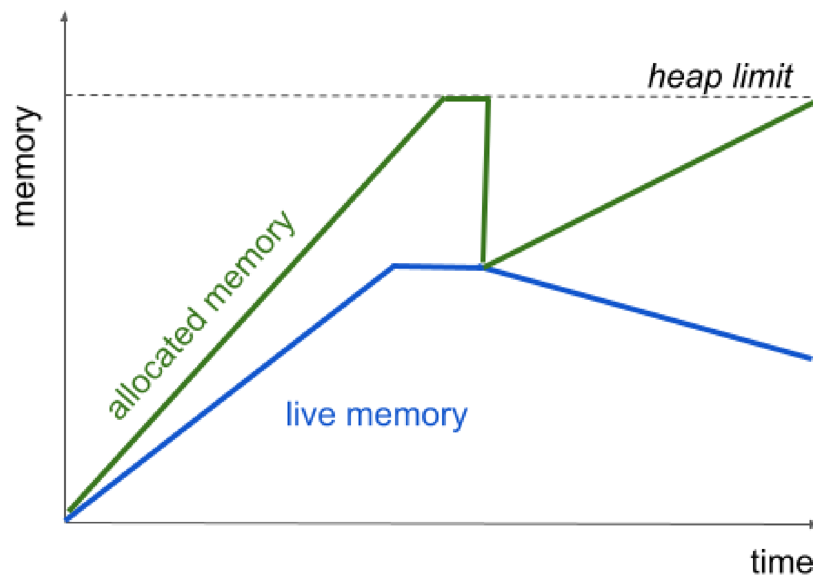
4.2 GC (4 pts)

Suppose a non-generational garbage collector is triggered when allocated memory reaches a heap limit. This collector uses a stop-the-world approach during collection.

Complete the memory/time profile below by adding a possible curve for the amount of memory that is currently allocated by this system. Your completed chart should include exactly one GC cycle.



A possible completion is shown below:



To be accepted, your completion must have the following properties:

- It must consist of 2 monotonically increasing sections - one before, and one after the GC cycle. Without garbage collection the amount of allocated memory steadily increases. Neither section has to be strictly monotonic.
- The curve must be at or above the live memory curve. By definition, only a subset of allocated memory is live.
- The curve must hit the heap limit to trigger a GC. Stated in the problem.

- The GC itself should take a non-zero amount of time. A non-generational collector using a stop-the-world approach requires at least some time for its mark phase.
- After the GC has finished, the curve must decrease to touch (or at least closely approach) the live memory curve - this represents that allocated memory was freed at the end of the GC cycle. In the ideal case, the 2 curves will touch.

5 Virtualization (8 pts)

Determine whether the following statements related to virtualization and containers are true or false.

(a) Both containers and virtual machines are common options offered by cloud providers today.

☑ true / ☐ false.

(b) A container image is a set of programs, data, and instructions on how to spin up a container instance.

☑ true / ☐ false.

(c) When using containers, application providers must package a separate version of their container image for each OS on which docker runs (i.e. Windows, Linux, OSX) and then publish it at the corresponding container registry.

☐ true / ☑ false.

The environment inside the container is always Linux-compatible, independent of where the container is executing.

(d) Amazon AWS's term of service prohibits users from mining cryptocurrencies without prior approval because letting users do so could affect AWS's ability to offer competitive pricing and/or make a profit.

☑ true / ☐ false.

(e) As per Popek and Goldberg's principles, virtual machines use direct execution but containers cannot.

☐ true / ☑ false.

Containers are user process so they, too, use direct execution.

(f) The use of virtual machines (as opposed to containers) is necessary if an application requires not only a specific Linux distribution, but also a specific release of the Linux kernel to run successfully.

☑ true / ☐ false.

(g) Containers can provide separate namespaces for the filesystem, network ports, and process ids. Virtual machines are required if, in addition, user ids and mount points need to be virtualized as well.

☐ true / ☑ false.

No, containers have their own user id and mount namespaces as well.

(h) Recent versions of x86-based microprocessors include hardware technology to support virtualization whose principles were originally developed in the 1970's.

☑ true / ☐ false.