

CS 3214 Fall 2022 Final Exam Solutions

December 15, 2022

1 Networking (28 pts)

1.1 Are you smarter than the CS3214 AI TA? (22 pts)

A CS3214 student developed a CS3214 AI TA based on GPT-3. We asked the AI TA a few questions about networking-related topics.

If the CS3214 AI TA answered the question correctly, check the box next to **Correct**. If not, check the **Not correct** box and explain what it got wrong (or what's wrong with the question). Note that the question may contain errors and note that GPT3 sometimes tends to answer a question that wasn't asked; both cases count as incorrect for the purposes of this question and require a correction.

(a) Q. Does every HTTP/1.1 request have a request body?

A. AI's answer:

No, not all HTTP/1.1 request have a request body. Requests such as GET and HEAD requests do not have a request body, while requests such as POST and PUT do have a request body.

Correct

(b) Q. When would an HTTP client know that it won't need to send any more requests over an existing connection?

A. AI's answer:

An HTTP client will know that it won't need to send any more requests over an existing connection when it receives a response with a Connection: close header from the server. This header indicates that the server is closing the connection and that the client should not attempt to send any more requests to it.

Not correct: The clients knows that it won't need to send any more requests based on context - for instance, if it has loaded all referenced assets in a webpage, or if the user requested only a single object as when running curl. Saying that it depends on whether the server sends a Connection: close header is like saying you know you won't need groceries because the store is closed.

(c) Q. What is required to forge a JWT token without providing username and password?

A. AI's answer:

It is not possible to forge a JWT token without providing the username and password. JWT tokens are cryptographically signed, so any attempt to modify the token will invalidate the signature and the token will be rejected.

Correct

(d) Q. What does the Path attribute in a cookie do?

A. AI's answer:

The Path attribute in a cookie is used to specify the URL path that the cookie should be sent with. This allows the server to restrict the cookie to certain paths on the website. For example, if the Path attribute is set to `"/private"`, then the cookie will only be sent with requests to URLs that start with `"/private"`.

Correct

(e) Q. How many state machines does a bidirectional TCP connection maintain?

A. AI's answer:

A bidirectional TCP connection maintains two state machines, one for each direction of communication.

Correct

(f) Q. What block cipher does TCP use by default?

A. AI's answer:

TCP does not use a block cipher by default. It uses a stream cipher, such as RC4, to encrypt data.

Not correct: TCP does not provide encryption as part of its services.

(g) Q. How does TCP demultiplex incoming packets?

A. AI's answer:

TCP demultiplexing is the process of taking incoming packets and routing them to the appropriate application. This is done by examining the source and destination IP addresses, source and destination port numbers, and the protocol field in the IP header. The combination of these fields is used to identify the application that the packet is intended for.

Correct

(h) Q. Does UDP's retransmission timeout depend on the roundtrip time (RTT)?

A. AI's answer:

Yes, UDP's retransmission timeout is dependent on the roundtrip time (RTT). The RTT is the time it takes for a packet to travel from the sender to the receiver and back again. The retransmission timeout is the amount of time the sender will wait for an acknowledgement before retransmitting the packet. The RTT is used to calculate the retransmission timeout so that the sender can adjust the timeout based on the current network conditions.

Not correct: UDP is an unreliable data transmission protocol that does not retransmit packets. (TCP uses the RTT to determine what retransmission timeout to use.)

(i) Q. In what order does `getaddrinfo()` return addresses to which my server can bind?

A. AI's answer:

The `getaddrinfo` function returns a list of addresses in the order in which they should be tried. Generally, it will return IPv6 addresses first, followed by IPv4 addresses.

☑ Not correct: Unfortunately, this is not how it works as you learned in p4. (It is true for when a client uses `getaddrinfo` to find the address of a server to connect to, but unfortunately not for a server that must find the address(es) to which to bind to accept clients.)

(j) Q. Show me how to parse cookie headers.

A. AI's answer:

Parsing cookie headers is a relatively simple process. This first step is to split the cookie header into individual cookie strings. This can be done by splitting the cookie header on the semicolon character. Once the individual cookie strings have been obtained, each cookie string can be split on the equals character to obtain the cookie name and value. Finally, the cookie name and value can be stored in a data structure such as a hash table or dictionary.

☑ Correct

(k) Q. What is the difference between `read()` and `recv()`

A. AI's answer:

The `read()` system call reads data from a file descriptor, while the `recv()` system call reads data from a socket. The `read()` system call is used to read data from a file, while the `recv()` system call is used to read data from a socket. The `read()` system call is a blocking system call, while the `recv()` system call is a non-blocking system call.

☑ Not correct: `read()` can be used uniformly on file descriptors that refer to files or sockets. Also, `recv()` is not a non-blocking system call - as you know, your p4 used `recv()` in blocking mode. Whether the call is blocking or not doesn't depend on whether `read()` or `recv()` are used, it depends on whether the file descriptor was placed in non-blocking mode or not. (*Needed to mention only one of these facts.*)

(l) This one is for extra credit:

The screenshot shows a chat interface with a question input field and a submit button. The question is: "What is Lauer and Needham's duality argument?". The answer provided is: "Lauer and Needham's duality argument is a philosophical argument that states that the two fundamental aspects of computing, namely hardware and software, are inextricably linked. The argument states that hardware and software are two sides of the same coin, and that neither can exist without the other. This argument has been used to explain why software engineering is so difficult, and why hardware and software must be designed together."

☑ Not correct: GPT-3 was really eager to please with this one even though it clearly hadn't read the paper, causing it to hallucinate. Lauer and Needham's duality argument is about the duality of message-passing vs processes, or, expressed in modern lingo, events vs. threads.

1.2 Shortcomings of HTTP/1.1 (6 pts)

In project 4, you developed a simple HTTP/1.1 server. During the last decade, newer versions of HTTP have been developed, such as HTTP/2 and HTTP/3 (QUIC). Provide 2 examples of shortcomings of HTTP/1.1 that are addressed by these new protocols:

Any of the innovations introduced in HTTP/2 or HTTP/3 address shortcomings, including

- Head-of-Line Blocking on the TCP connection because objects must be returned in the order in which they were requested (solved by HTTP/2 using framing)
- Head-of-Line Blocking due to TCP retransmission (solved by HTTP/3)
- Provides incentives for clients to create multiple independent TCP connections which consume network and host resources. (Addressed in both HTTP/2 and /3).
- Large header overhead (solved by header compression)
- Inability for server to push objects (addressed by separate mechanisms in HTTP/2 and /3)
- Needing multiple RTTs for TCP + TLS handshake before request can be sent (addressed by HTTP/3).
- Lack of mandatory encryption.

Note: $11 \times 2 = 22$ pts were achievable, plus 2 pts of extra credit. If the 3214 AI was correct and correct was checked, 2 pts. If the 3214 AI was incorrect, 1 pt for checking not correct and 1 pt for a correct explanation of why.

2 Virtual Memory (20 pts)

Consider the following C program:

```

1  #define _GNU_SOURCE 1
2  #include <sys/types.h>
3  #include <sys/mman.h>
4  #include <sys/wait.h>
5  #include <fcntl.h>
6  #include <stdlib.h>
7  #include <unistd.h>
8  #include <stdint.h>
9  #include <stdio.h>
10
11 #define PGSIZE 4096
12 #define ROUND_DOWN_TO_PAGE(x) (((uintptr_t)x) & ~(PGSIZE-1))
13 #define N 4
14 static char global[N*PGSIZE] __attribute__((__aligned__(PGSIZE)));
15
16 int main(int ac, char *av[])
17 {
18     void *addr = mmap((void*)0x8888888000, N * PGSIZE, PROT_READ | PROT_WRITE,
19                     MAP_ANONYMOUS | MAP_PRIVATE | MAP_FIXED, -1, 0);
20
21     int *as_int = addr;
22     as_int[1200] = 5;
23     printf("echo status of pages of anonymous mmap\n");
24     printf("./pagemap3 %d %p %p\n", getpid(), addr, addr + N*PGSIZE);
25
26     char *exe = mmap(NULL, PGSIZE, PROT_READ, MAP_SHARED,
27                    open("/proc/self/exe", O_RDONLY), 0);
28     printf("echo 'exe[1..3] = ' %c%c%c\n", exe[1], exe[2], exe[3]);
29     printf("echo status of pages of mapped /proc/self/exe:\n");

```

```

30     printf("./pagemap3 %d %p %p\n", getpid(), exe, exe + PGSIZE);
31
32     global[10000] = 'x';
33     printf("echo status of pages of global array\n");
34     printf("./pagemap3 %d %p %p\n", getpid(), global, global + N*PGSIZE);
35
36     void *mainpage = (void *) ROUND_DOWN_TO_PAGE(main);
37     printf("echo status of main\n");
38     printf("./pagemap3 %d %p %p\n", getpid(), mainpage, mainpage + PGSIZE);
39
40     void *small = (void*) ROUND_DOWN_TO_PAGE(malloc(4));
41     printf("echo status of small malloc\n");
42     printf("./pagemap3 %d %p %p\n", getpid(), small, small + PGSIZE);
43
44     sleep(10000);
45     return 0;
46 }

```

When run on a 64-bit Linux machine where `sizeof(int) == 4`, it outputs the following shell script

```

1 echo status of pages of anonymous mmap
2 ./pagemap3 1284207 0x8888888000 0x888888c000
3 echo "exe[1..3] = " ___
4 echo status of pages of mapped /proc/self/exe:
5 ./pagemap3 1284207 0x7f05998d0000 0x7f05998d1000
6 echo status of pages of global array
7 ./pagemap3 1284207 0x55a884614000 0x55a884618000
8 echo status of main
9 ./pagemap3 1284207 0x55a88460f000 0x55a884610000
10 echo status of small malloc
11 ./pagemap3 1284207 0x55a8853ae000 0x55a8853af000

```

This shell script invokes a tool `pagemap3` which uses a kernel interface to obtain information about a range of virtual pages in a running process, specifically the following three attributes:

- `is_a_file` is 1 if the page is backed by a file, 0 otherwise
- `is_swapped_out` is 1 if the page was evicted from its frame into swap space, 0 otherwise
- `is_present` is 1 if the page is present (aka resident) in physical memory, 0 otherwise

Predict the output of this shell script! **Complete the output below by filling in the correct digit or letter for each underscore.** This requires filling in 33 single digits (0 or 1).

```

status of pages of anonymous mmap
0x8888888000      is_a_file _ is_swapped_out _ is_present _
0x8888889000      is_a_file _ is_swapped_out _ is_present _
0x888888a000      is_a_file _ is_swapped_out _ is_present _
0x888888b000      is_a_file _ is_swapped_out _ is_present _

exe[1..3] = ___
status of pages of mapped /proc/self/exe:
0x7f05998d0000    is_a_file _ is_swapped_out _ is_present _

```

```

status of pages of global array
0x55a884614000    is_a_file _ is_swapped_out _ is_present _
0x55a884615000    is_a_file _ is_swapped_out _ is_present _
0x55a884616000    is_a_file _ is_swapped_out _ is_present _
0x55a884617000    is_a_file _ is_swapped_out _ is_present _

status of main
0x55a88460f000    is_a_file _ is_swapped_out _ is_present _

status of small malloc
0x55a8853ae000    is_a_file _ is_swapped_out _ is_present _

```

Assume a strictly on-demand paged system that it operating with plenty of free RAM. The `pagemap3` utility outputs the virtual address of the page in question in the left column.

For extra points, predict the three letters after `exe[1..3]`, which are magic characters inside a header that denotes the type of the file in question; it is a 3-letter acronym.

Hint: `/proc/self/exe` is a symbolic link to the current process's executable.

```

status of pages of anonymous mmap
0x8888888000      is_a_file 0 is_swapped_out 0 is_present 0
0x8888889000      is_a_file 0 is_swapped_out 0 is_present 1
0x888888a000      is_a_file 0 is_swapped_out 0 is_present 0
0x888888b000      is_a_file 0 is_swapped_out 0 is_present 0

exe[1..3] = ELF
status of pages of mapped /proc/self/exe:
0x7f05998d0000    is_a_file 1 is_swapped_out 0 is_present 1

status of pages of global array
0x55a884614000    is_a_file 0 is_swapped_out 0 is_present 0
0x55a884615000    is_a_file 0 is_swapped_out 0 is_present 0
0x55a884616000    is_a_file 0 is_swapped_out 0 is_present 1
0x55a884617000    is_a_file 0 is_swapped_out 0 is_present 0

status of main
0x55a88460f000    is_a_file 1 is_swapped_out 0 is_present 1

status of small malloc
0x55a8853ae000    is_a_file 0 is_swapped_out 0 is_present 1

```

- None of the pages is swapped out since the problem stated the assumption of “plenty of free RAM.”
- Anonymous mmap'd memory is not file backed. `&as_int[1200]` is 4,800 bytes above the base address, which causes page 0x888888900 to be touched and become present.
- The mmap'd file at `/proc/self/exe` is of course a file, and since we just loaded 3 bytes from this address range, it is also present.
- The first four bytes contain the ELF header, which in fact contains the letters ‘E’, ‘L’, and ‘F’.

- The global array is part of the BSS segment, which is not stored in any file. We touch offset 10 000, which is on the third page since the array is page-aligned.
- The code of the `main` function is part of the executable, which is mapped as a file. This is possible because the text portion of an executable is read-only and thus can be shared if multiple processes run the same executable. Since there is code running inside `main()`, the processor must have fetched instructions from this page, which required bringing it into physical memory, thus making it present (otherwise, `main()` could not have run.)
- Small `malloc()` calls are satisfied from a heap that is managed similar to what you implemented in p3. The `malloc()` implementation thus will write block headers and footers, causing the page in which the object is located to be present (if it wasn't already due to other operations by the memory allocator, such as manipulating free lists and writing list elements). Therefore, the page is present even though no data was written to the allocated portion yet. This would not necessarily be true had the allocated block been larger and spanned multiple pages.

We awarded points by column.

- `is_a_file` column: 8 pts max. Deduct 1 per wrong answer, capping at 0.
- `is_swapped_out` column: 4 pts max. Deduct 1 per wrong answer, capping at 0.
- `is_present` column: 8 pts max. Deduct 1 per wrong answer, capping at 0.
- +2 extra points for ELF

3 Dynamic Memory Management (18 pts)

Consider the following program, which uses Linux's default implementation of `malloc()` and `free()`.

```

1  #include <stdlib.h>
2  #include <malloc.h>
3  #include <stdio.h>
4
5  int
6  main()
7  {
8      void *a0 = malloc(96);
9      void *a1 = malloc(96);
10     void *a2 = malloc(96);
11     free(a1);
12     void *a3 = malloc(192);
13     free(a2);
14     void *a4 = malloc(160);
15     void *a5 = malloc(96);
16     void *a6 = malloc(96);
17
18     printf("a0 = malloc(96) -> %p \n ", a0);
19     printf("a1 = malloc(96) -> %p = a0 + %d \n ", a1, a1 - a0);
20     printf("a2 = malloc(96) -> %p = a0 + %d \n ", a2, a2 - a0);
21     printf("free(a1=%p) \n ", a1);
22     printf("a3 = malloc(192) -> %p = a0 + %d \n ", a3, a3 - a0);
23     printf("free(a2=%p) \n ", a2);

```

```

24     printf("a4 = malloc(160) -> %p = a0 + %d \n ", a4, a4 - a0);
25     printf("a5 = malloc(96)  -> %p = a0 + %d \n ", a5, a5 - a0);
26     printf("a6 = malloc(96)  -> %p = a0 + %d \n ", a6, a6 - a0);
27 }

```

when run on our rlogin cluster with address space layout randomization (ASLR) turned off, we obtain this output:

```

a0 = malloc(96)  -> 0x6022a0
a1 = malloc(96)  -> 0x602310 = a0 + 112
a2 = malloc(96)  -> 0x602380 = a0 + 224
free(a1=0x602310)
a3 = malloc(192) -> 0x6023f0 = a0 + 336
free(a2=0x602380)
a4 = malloc(160) -> 0x6024c0 = a0 + 544
a5 = malloc(96)  -> 0x602380 = a0 + 224
a6 = malloc(96)  -> 0x602310 = a0 + 112

```

- (a) (3 pts) How many bytes are lost to internal fragmentation in block a0?

112 - 96 = 16 bytes appear to be lost to internal fragmentation (assuming that the block the allocator allocated runs from a0 - x to a1 - x where x is the boundary tag header size).

- (b) (6 pts) Based on the addresses returned for a4, a5, and a6, what can you conclude about the policies pursued by this allocator? Is the observed behavior the same as what your p3 allocator would have returned? Explain why or why not.

Blocks a1 and a2 (each holding 96 bytes of payload) were adjacent, but even after both are freed, the resulting area in memory is not reused to satisfy an allocation request of size 160. Most p3 implementations would have immediately coalesced these 2 freed blocks and used them to satisfy the 160 byte request - for instance, because this block would have been found in the same or next larger size class. However, GNU's malloc() keeps dedicated free lists for small blocks it does not coalesce, so-called "fast bins," which we observe here in action.

- (c) (3 pts) Does this allocator use a LIFO or a FIFO insertion policy for its free lists?

LIFO / FIFO

because a5 reuses the block a2 previously used, which was most recently freed.

- (d) (6 pts) Now consider what happens when we increase the requested size of each block by a factor of 100. We now obtain this output:

```

a0 = malloc(9600) -> 0x6022a0
a1 = malloc(9600) -> 0x604830 = a0 + 9616
a2 = malloc(9600) -> 0x606dc0 = a0 + 19232
free(a1=0x604830)
a3 = malloc(19200) -> 0x609350 = a0 + 28848
free(a2=0x606dc0)
a4 = malloc(16000) -> 0x604830 = a0 + 9616
a5 = malloc(9600)  -> 0x60de60 = a0 + 48064
a6 = malloc(9600)  -> 0x6103f0 = a0 + 57680

```

Describe what changed compared to the smaller size case.

Now blocks a1 and a2 were immediately coalesced and reused for the allocation of 16 000 bytes for a4. This then caused the allocations of blocks a5 and a6 to be allocated after them, likely expanding

the heap. This shows that GNU malloc() does not delay coalescing for large blocks. This behavior is probably identical to most p3 implementations.

4 Automatic Memory Management (18 pts)

4.1 Understanding Object Reachability (9 pts)

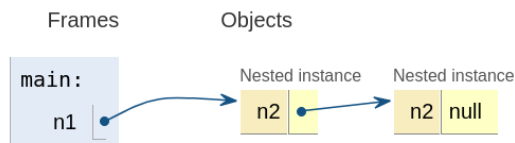
Consider the following Java program:

```

1 public class YourClassNameHere {
2     static class Nested {
3         Nested n2;
4         Nested(boolean nest) {
5             if (nest)
6                 n2 = new Nested(false);
7         }
8     }
9
10    public static void main(String[] args) {
11        Nested n1 = new Nested(true);
12        Nested n3 = n1.n2;
13        n1 = null;
14    }
15 }

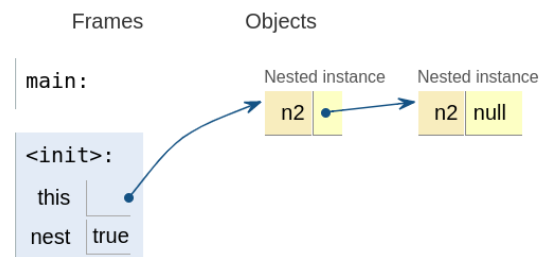
```

The following drawings show the structure of the live heap at three different points of execution of this program. Identify those points



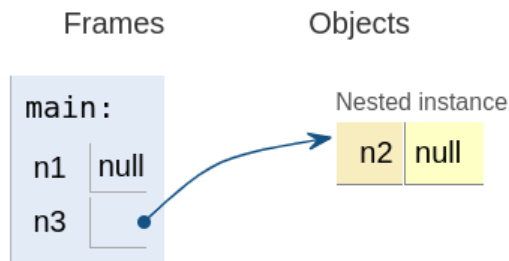
(a) (3 pts)

This situation arises after line 11 has executed.



(b) (3 pts)

This situation arises after line 6 has executed.



(c) (3 pts)

This situation arises after line 13 has executed.

4.2 Memory Leak in Python (9 pts)

On the CS3214 forum, a student shared a link to an article published on the website “Geeks for Geeks” that discusses a “memory leak” in Python. The author of this article analyzed the following program:

```

1 import requests
2 import gc
3
4 def call():
5     response = requests.get('https://google.com')
6     print("Status code",response.status_code)
7     return
8
9 def main():
10    print("No.of tracked objects before calling get method")
11    print(len( gc.get_objects() ) )
12    call()
13    print("No.of tracked objects after calling get method")
14    print(len( gc.get_objects() ) )
15
16 if __name__ == "__main__":
17    main()

```

This program uses the `gc.get_objects()` method, which returns the number of currently allocated but not yet freed objects. When the author ran it, this program output:

```

No.of tracked objects before calling get method
16071
Status code 200
No.of tracked objects after calling get method
16158

```

They concluded that because the “returned object count (after calling get) is greater, (...) there is a memory leak.”

However, when they added a call to invoke Python’s garbage collector after line 12, they observed that the number of reported objects went down to 15954. They concluded that they were able to “Fix (the) Memory leak” by manually calling the `gc.collect()` method (which triggers a garbage collection in Python).

(a) (3 pts) What could account for the larger number of allocated objects observed on line 14 compared to line 11?

Languages such as Python that use automatic memory management allocate temporary, short-lived objects as they run. Any temporary objects allocated during the call to `requests.get()` may not have been garbage collected on line 14 and thus show up in the statistics of currently allocated but not yet freed objects. In addition, a library like `requests` may allocate singleton objects on a first call, and it may also allocate and retain certain objects it intends to reuse in the future, such as pooled connections.

- (b) (3 pts) Is the author of this article correct in concluding the existence of a leak inside the Python `requests` library (invoked on line 5), based on their observations? (Justify your answer.)

The author observes that the number of currently allocated objects goes down after explicitly invoking the collector. This is the expected behavior of garbage collection; it is not a leak. A leak would be if the number of allocated objects did not go down after collection.

- (c) (3 pts) Can memory leaks in a language such as Python be fixed by invoking the garbage collector? (Briefly say why or why not.)

No. Leaks in garbage collected languages like Python occur when still reachable objects aren't accessed in the future. If the garbage collector is able to free objects, they are not reachable, and thus not part of a leak.

5 Virtualization (16 pts)

To achieve concurrent execution, application developers have access to a wide spectrum of approaches. This question asks you to show your computer systems knowledge regarding a range of these approaches.

Answer the questions from the perspective of the programmer creating applications and address the default case (not setups that may be achievable using special-purpose APIs). Assume multiple instances of each entity, colocated as closely as possible (e.g., consider multiple threads within one process, not multiple threads running on different physical machines, consider multiple processes in a container, multiple containers in a virtual machine, etc.)

	Threads	Processes	Containers	Virtual machines	Physical machines
Provide independent control flows	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Share all file descriptors	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Share a filesystem namespace, e.g. <code>/bin</code>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Share a virtual address space	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Share a namespace of user ids	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Share a kernel	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Share a physical machine	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Provide the strongest level of isolation	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

- All forms provide independent control flows.
- Only threads within a process share all file descriptors and a virtual address space. Any of the others do not.
- Threads and processes (within a container, virtual machine, or physical machine) share a filesystem namespace and user id namespace. Different containers can have different filesystem and user id namespaces.
- Threads within a process and processes within a container and containers within a virtual or physical machines share a kernel. Virtual or physical machines do not.

- Threads within a process, processes within a container, containers within a virtual machine, and virtual machines running on a physical machine share the physical machine on which they're running.
- Physical machines provides the strongest level of isolation.

All 40 entries correct gave 16 points. We subtracted 1 pt per wrong answer, capping at 0.