CS 3214 Spring 2023 Final Exam Solutions

May 10, 2023

1 Networking (27 pts)

1.1 Know Your Internet (9 pts)

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

(a) "Forwarding" refers to the local decision of choosing the next hop to which to send an incoming packet, whereas "Routing" refers to protocols that provide the necessary information on how to forward.

[✓ true] / ☐ false.

(b) On a 100Gbps intercontinental fiber link, the propagation delay is orders of magnitude higher than the transmission delay for a packet of $1,500$ bytes of data.

[✓ true] / ☐ false.

(c) The IPv4 and IPv6 protocols are designed to provide a "best effort" packet transmission service, but they do not guarantee reliable transfer.

[✓ true] / ☐ false.

(d) The term "pipelining" refers to a property of a transport or application layer protocol where multiple packets or requests may be sent before a response to or acknowledgment of the first request or packet was received.

[✓ true] / ☐ false.

(e) To create a TCP connection to a peer across the Internet, the initiating client must send a connection request message that is processed by each node along the path in order to reserve per-hop bandwidth for the connection.

☐ true / [✓ false].

(f) The basic semantics of the HTTP protocol we use today has been largely unchanged since the 1990's, but the way in which its semantics is mapped to underlying transport layer connections has evolved.

[✓ true] / ☐ false.

(g) If an attacker gains access to a user agent's cookie storage ("cookie jar") they may be able to impersonate the user when accessing HTTP-based online sites or services.

[✓ true] / ☐ false.

(h) In a REST API, transferred objects are typically represented using Hyper Text Markup Language (HTML).

☐ true / [✓ false].

(i) Transport protocols such as TCP or QUIC include a mechanism to protect the network from congestion when they detect packet loss from overflowing router queues.

[✓ true] / ☐ false.

1.2 HTTP Transactions (6 pts)

A CS3214 group was debugging their p4 implementation of a compliant HTTP/1.1 server implementing the `/api/login` API. In response to this request:

```
POST /api/login HTTP/1.1
Host: spruce:9998
User-Agent: curl/7.86.0
Accept: */*
Content-Type: application/json
Content-Length: 46

{"username":"user0", "password":"thepassword"}
```

Their server responded with:

```
HTTP/1.2 200 OK
Server: CS3214-Personal-Server
Set-Cookie: auth_token=elided.for.brevity; Path=/api/login; Max-Age=86400; SameSite=Lax; HttpOnly
Content-Type: text/plain; charset=utf8
Content-Length: 2

{"exp":1683328883,"iat":1683242483,"sub":"user0"}
```

List 3 mistakes in their response:[1]

a) (2 pts) Mistake 1: The protocol version should be `HTTP/1.1`, not `HTTP/1.2`

b) (2 pts) Mistake 2: The content length should be larger than 2.

c) (2 pts) Mistake 3: The content type should be `application/json`

d) Mistake 4: The `Path` attribute of the cookie should be `/`, not `/api/login`.

1.3 Concurrent Web Servers (4 pts)

An engineer wants to put an easter egg into a web application they are writing. They insert code so that, when a request containing a specific magic word arrives, in its request handling code, their application performs a synchronous (blocking) call to contact a video server in order to fetch a large cat video, then responds with this video instead. For this question, assume that bandwidth is plentiful and effects from network contention are negligible.

(a) (2 pts) Suppose the engineer is using a thread-based framework for their web application. What will happen to the performance of other clients concurrently accessing the web application if one client triggers the easter egg?

The thread handling the request will block retrieving the cat video, but the other threads will continue to progress as normal, with minimal impact on other clients.

(b) (2 pts) Suppose the engineer is using an single-threaded event-based framework for their web application. What will happen to the performance of other clients concurrently accessing the web application if one client triggers the easter egg?

[1]Note that the elision of the cookie value is for expository purposes and does not count as a mistake.

In this case, the thread would block the entire process, and thus block further processing of any clients in the event loop, leading to a loss of concurrency and increased latency/lack of service for other clients.

1.4 Effective Bandwidth (8 pts)

In this question we examine the effective bandwidth achievable across a link given the impact of protocol headers. For the purposes of this question, we assume a simplified version of all protocols including IP and UDP in which there is no fragmentation and/or reassembly; in other words, we assume the entire packet or datagram must fit in a single Ethernet frame. For the following calculations, since you don't have a calculator to use, it is OK to write a numeric expression instead of computing the final number.

Suppose a student is running tests across a 1Gbps Ethernet link connecting two computers. Suppose the maximum transmission unit (MTU) containing the Ethernet payload is 1500 bytes, and that the IP and UDP headers are 20 and 8 bytes, respectively. Additionally, assume Ethernet headers and trailers add an overhead of 38 bytes per frame on top of the 1500 MTU.

(a) (2 pts) What is the effective bandwidth (the portion of the available bandwidth not being used for headers) when using UDP directly across this link?

Of the available $1500 + 38$ bytes per frame we must subtract the additional bytes used for the IP and UDP headers, resulting in $\frac{1500-20-8}{1538} \times 1$ Gbps $\approx .96$ Gbps

(b) (2 pts) Suppose the student wants to implement a new protocol layer on top of UDP. The new protocol will have many features, captured by an extended 200 byte protocol header. When layering their new protocol atop UDP, what effective bandwidth can the student expect to achieve?

Similar to the above, we must subtract the additional bytes used for the new protocol header, resulting in $\frac{1500-200-20-8}{1538} \times 1$ Gbps $\approx .83$ Gbps

(c) (2 pts) The student learns about Jumbo Ethernet frames with an MTU of 9000 bytes that could potentially be used on their 1Gbps link. If this is the case, what effective bandwidth can they expect on this network?

Given the larger frame, we get $\frac{9000-200-20-8}{9038} \times 1$ Gbps $\approx .97$ Gbps

(d) (2 pts) Other than using Jumbo Ethernet frames, what could the student do to their protocol header to increase the effective bandwidth?

Anything that reduces the size of the protocol header will increase the effective bandwidth, including compression (such as header compression in HTTP/2), or rethinking the format or encoding to shrink the size.

2 Virtual Memory (18 pts)

2.1 Understanding Demand Paging (6 pts)

The `time(1)` utility runs a Linux program and then reports statistics about its execution. One statistic is the "maximum resident set size," which is the maximum amount of physical memory allocated by the OS to the process in its entirety. Note that this includes a bit of "system noise," leading to slight variations from run to run. It also includes a "base cost" incurred by the C library, the dynamic linker and loader, etc., which is roughly constant for each program.

Consider the following 3 programs:

prog1

```
#include <string.h>
#include <stdint.h>

int32_t b[4096*256];

int
main()
{
    memset(b, 0, sizeof b);
}
```

prog2

```
#include <string.h>
#include <stdint.h>

int32_t b[4096*1024];

int
main()
{
}
```

prog3

```
#include <string.h>
#include <stdlib.h>

int
main()
{
    const int SZ = 4096*512;
    char *b = malloc(SZ);
    memset(b, 0, SZ);
}
```

when run with `/usr/bin/time -f "RSS= %MKB" prog` the following outputs were obtained. Check next to each output which program produced it:

a) `RSS= 3360KB` produced by (check one) ☐ prog1 ☐ prog2 [✓ prog3]

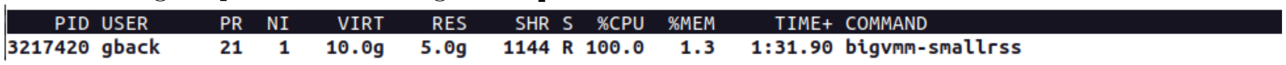
b) `RSS= 5444KB` produced by (check one) [✓ prog1] ☐ prog2 ☐ prog3

c) `RSS= 1344KB` produced by (check one) ☐ prog1 [✓ prog2] ☐ prog3

prog2 does not touch any physical memory, so its RSS will provide the baseline cost. Since `sizeof(int32_t) == 4`, prog1 touches about $4096 \cdot 256 \cdot 4$ bytes, or about 4,096KB above the baseline cost. prog3 touches about $4096 \cdot 512$ bytes or about 2,048KB above the baseline cost. We should note that these numbers, particularly the baseline cost, are overestimates in the sense that they include memory that is in fact shared with other processes. Starting an empty process like prog2 does not incur an incremental cost of 1,344KB.

2.2 Virtual Memory Consumption (4 pts)

Dr. Back wrote a tiny C program `bigvmm-smallrss.c` that, after being started in the background, produces the following output when running the `top` command:



```
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
3217420 gback     21   1   10.0g   5.0g   1144 R 100.0   1.3   1:31.90 bigvmm-smallrss
```

Reproduce the main C program; focus on the columns labeled `VIRT` (total amount of virtual memory addresses allocated), `RES` (total amount of physical memory used), and `S` (process state as per Linux conventions). Include headers etc. are not necessary.

```
#include <stdlib.h>
#include <string.h>

int
main()
{
    memset(malloc(10L*1024*1024*1024), 0, 5L*1024*1024*1024);
    for(;;)
        ;
}
```

A correct answer needed to include

- A virtual memory allocation on the order of 10GB, which could be a call to malloc(), or a direct call to mmap() or even sbrk(), and possibly even a global variable definition like in the previous examples.
- An operation that touches 5GB, or half of that memory - such as memset(), or a loop with a read or write access. It is sufficient to just touch one byte in each page.
- A concluding CPU-bound loop to produce the `R` state and CPU consumption.

2.3 Observing `mmap()` (8 pts)

Consider the following program `mmap.c`:

```
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <assert.h>

int
main(int ac, char *av[])
{
    int fd = open(av[1], O_RDWR);
    assert (fd != -1);

    struct stat st;
    assert (fstat(fd, &st) == 0);
    off_t filesize = st.st_size;

    void *addr = mmap(NULL, filesize, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED) {
        perror("mmap"); exit(-1);
    }

    assert ('q' - 'Q' == 32); // hint
    // access file data like memory
    char *start = addr, *end = addr + filesize;
    while (start < end) {
        if ('A' <= *start && *start <= 'Z')
            *start = *start | 0x20;
        start++;
    }

    execvp("cat", av);
    return 0;
}
```

Now consider the following sequence of Unix shell commands:

```
$ gcc -Wall mmap.c -o mmap
$ echo 'This is a TEST.' > testfile
$ ./mmap testfile
```

Predict what the user will see on their terminal as a result:

The user will see:

```
this is a test.
```

The program maps the file whose names is given as a first argument into memory, then replaces all uppercase with their corresponding lowercase characters, then replacing the current program with the `cat` utility that outputs the changed file to its standard output stream which in this case is the terminal.

3 Dynamic Memory Management (22 pts)

3.1 Observing `malloc()` (16 pts)

Consider the following program, which uses `malloc()` and `free()` and outputs the addresses it obtains from the memory allocator.

```c
#include <stdlib.h>
#include <stdio.h>

int main()
{
    // to avoid dealing with large numerical addresses,
    // we will output all addresses relative to this point
    // of reference
    void *ref = malloc(128);
    void *a[3] = { NULL, NULL, NULL };

    printf("ref = malloc(128)  -> %p\n", ref);
    for (int i = 0; i < 3; i++) {
        if (i > 0) {
            for (int j = 0; j < 3; j++) {
                printf("free(a[%d] = %ld)\n", j, a[j]-ref);
                free(a[j]);
            }
        }
        for (int j = 0; j < 3; j++) {
            a[j] = malloc(128);
            printf("a[%d] = malloc(128)  -> %ld\n", j, a[j]-ref);
        }
    }
    // printf("malloc(127)  -> %ld\n", malloc(127)-ref);
    free(ref);
}
```

The Hoard is a memory allocator developed by Emery Berger in 2003. When the program above is run such that the Hoard's implementations of `malloc` and `free` are used instead of the GNU C library's, we obtain this output:

```
Using the Hoard memory allocator (http://www.hoard.org), version 3.13.0
ref = malloc(128)  -> 0x14a61fa00070
```

```
a[0] = malloc(128)  -> 128
a[1] = malloc(128)  -> 256
a[2] = malloc(128)  -> 384
free(a[0] = 128)
free(a[1] = 256)
free(a[2] = 384)
a[0] = malloc(128)  -> 384
a[1] = malloc(128)  -> 256
a[2] = malloc(128)  -> 128
free(a[0] = 384)
free(a[1] = 256)
free(a[2] = 128)
a[0] = malloc(128)  -> 128
a[1] = malloc(128)  -> 256
a[2] = malloc(128)  -> 384
```

(a) (3 pts) How many bytes are lost to internal fragmentation in block `a[0]`?

Amazingly, 0 bytes. Blocks ref, a[0], a[1] are back to back with no space lost to internal fragmentation. They all contain 128 bytes of payload but are 128 bytes spaced apart.

(b) (3 pts) Can you determine whether this allocator uses Knuth's boundary tag header technique discussed in class? Say why or why not.

The boundary tag header technique requires placing a header (and possibly footer) directly before the block's payload - since there is no space here, the technique could not have possibly been used.

Instead, the Hoard uses a segregated technique with same-sized blocks in which metadata such as the block size is stored in an aligned superblock header whose location can be found using virtual memory address arithmetic. (For instance, by rounding down to the nearest multiple of 256KB using bit operations.)

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

[✓ LIFO] / ☐ FIFO / ☐ Cannot determine

The block freed with the first `free(a[2])` statement is returned by the subsequent `malloc(128)` call.

(d) (4 pts) If we use valgrind's memory leak detection facilities and execute `valgrind --leak-check=full ./mallochoard`, we obtain this output

```
==39997== LEAK SUMMARY:
==39997==    definitely lost: ____ bytes in __ blocks
==39997==    indirectly lost: ____ bytes in __ blocks
==39997==      possibly lost: 0 bytes in 0 blocks
==39997==    still reachable: ____ bytes in __ blocks
==39997==         suppressed: 0 bytes in 0 blocks
==39997==
```

Fill in the 6 blanks. Remember that valgrind traces the live heap after `main()` has returned, which means that all local variables defined in `main()` have gone out of scope.

```
==39997== LEAK SUMMARY:
==39997==    definitely lost: 384 bytes in 3 blocks
==39997==    indirectly lost: 0 bytes in 0 blocks
==39997==      possibly lost: 0 bytes in 0 blocks
```

```
==39997==    still reachable: 0 bytes in 0 blocks
==39997==         suppressed: 0 bytes in 0 blocks
==39997==
```

Note that `ref` was explicitly freed before returning from main(), but the last 3 blocks allocated were not. The only pointers to those blocks were stored in local variables in main() which as the problem reminded you had already gone out of scope, thus nothing is still reachable at this point. There are also no pointers stored on the heap, thus nothing is indirectly lost.

As a reminder, using valgrind in this mode is simply for diagnostic purposes, these blocks of memory aren't actually "leaked" since the OS will reclaim any memory this process used upon exit().

(e) (3 pts) Suppose we uncomment line 25 and run the program using the Hoard (and without valgrind), predict what the print statement on line 25 would output for the relative offset of the address returned by `malloc(127)`.

As a compliant allocator, the Hoard has to ensure alignment, which implies rounding up the request to a suitable multiple of the required alignment - in this case, 128. From parts a) and b) you could infer that the Hoard uses a segregated storage scheme that allocates objects of size 128 closely together. Thus, the most likely prediction is a block that's 512 bytes away from `ref`.

3.2 Fact about Dynamic Allocators (6 pts)

Determine if the following statements related to dynamic memory allocation are true or false. Assume a traditional `malloc()`, `free()`, `realloc()` API as described by POSIX.

(a) The amortized cost of freeing an object in a dynamic memory allocator can be described as being on the order of $O(n)$ where n is size of the remaining heap memory.

☐ true / [✓ false]. The amortized cost of a free() is typically constant $O(1)$.

(b) Memory allocators must be implemented in the C language.

☐ true / [✓ false]. They can be implemented in other languages, too, such as Assembly, C++, or even Rust.

(c) Optimized memory allocators typically reorder requests, e.g., execute subsequent `malloc()` calls before ones that were issued earlier if doing so reduces fragmentation.

☐ true / [✓ false]. Reordering requests is not possible - allocators must respond synchronously.

(d) Internal fragmentation is difficult to measure because it depends on future access patterns which may not be known.

☐ true / [✓ false]. Internal fragmentation is easy to measure, it's external fragmentation that's hard.

(e) Memory allocators perform internal consistency checks that can help users detect memory usage errors in production when doing so does not cause significant performance overhead.

[✓ true] / ☐ false.

(f) Memory allocators may move blocks of allocated memory only as part of a `realloc()` call.

[✓ true] / ☐ false.

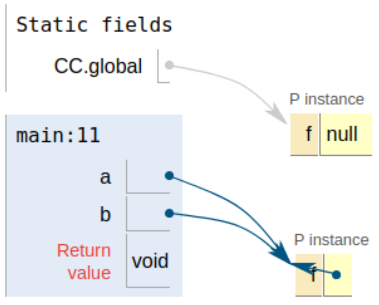
4 Automatic Memory Management (16 pts)

4.1 Reachability Graphs (6 pts)

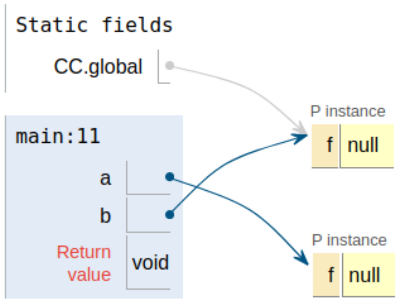
Consider the following Java program:

```
public class CC {
  static class P {
    P f;
  }
  static P global;
  public static void main(String[] args) {
    global = new P();
    P a = new P();
    P b = global;
    a.f = b;
  }
}
```

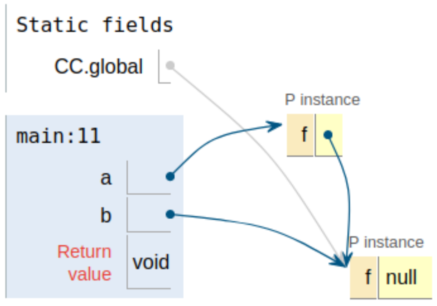
Which of the following graphs describes this program's reachability graph after executing the statements up until (and including) line 10? (Check the corresponding box.)



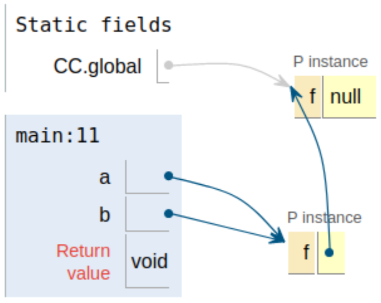
☐ Graph 1



☐ Graph 2



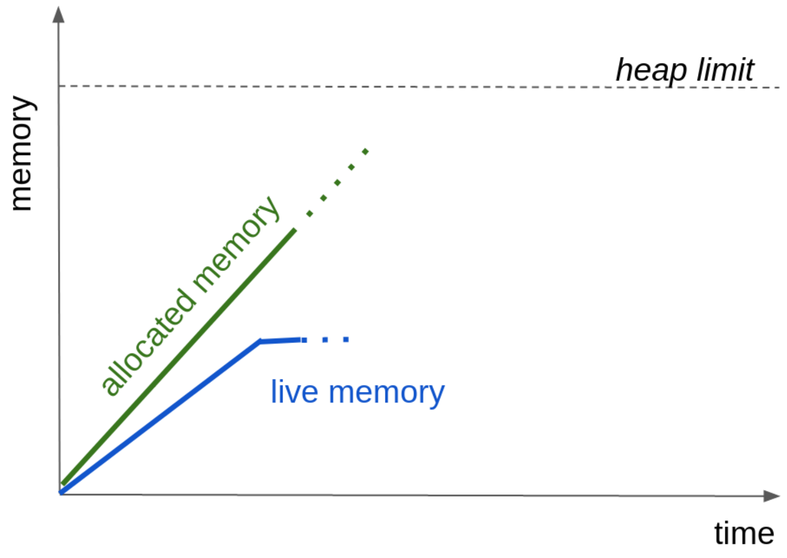
[✓ Graph 3]



☐ Graph 4

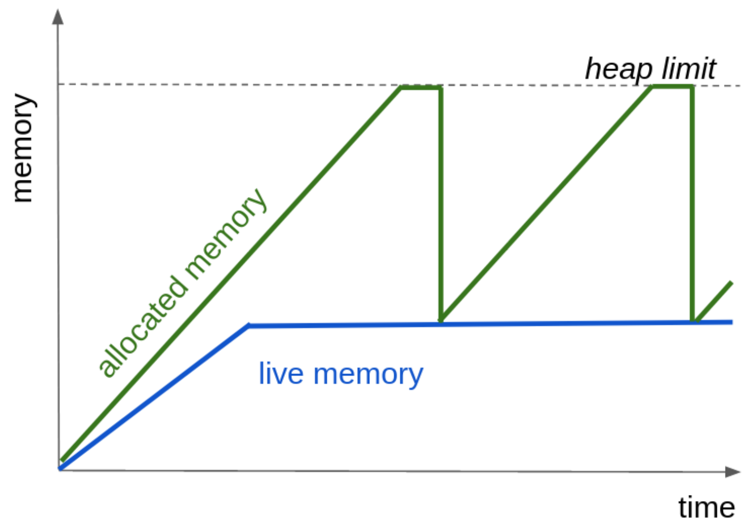
4.2 Understanding Memory/Time Profiles (10 pts)

Consider the following partial profile of an application running under the regime of an automatic memory manager:



(a) (4 pts) Assuming that the application continues to allocate memory at the same rate, and assuming that live memory stays at its current trajectory, show a possible completion of the diagram by continuing the allocated memory and live memory lines. (Draw into the diagram.)

The blue live memory line stays constant. The allocated memory line continues at its slope until garbage collection is triggered, at which point it would be reduced to live memory before climbing again.



Note to graders: we're just looking for the sawtooth pattern. Allowable modifications may include triggering GC via some threshold before the heap limit is exhausted, not showing the time to perform the GC itself (depends on exactly how the x-axis is measured and the specific mechanism used), and possibly not reclaiming all garbage immediately; but the basic idea should be there.

(b) (2 pts) Based on the information given, does this application exhibit symptoms of a memory leak? Justify your answer.

It does not exhibit symptoms of a memory leak since the live memory curve stays constant rather than increasing.

(c) (4 pts) Write a minimal program in any language that uses automatic memory management that would roughly exhibit above allocation profile.

Any program that allocates some memory, keeps it alive, and then continues allocating memory that is not kept alive. At an absolute minimum, something like this Java program:

```
byte []b = new byte[1024];
for (int i = 0; i < 100; i++)
    new byte[1024];  // not kept alive
```

5 Virtualization (17 pts)

5.1 Who's Job is it in Cloud Computing (7 pts)

In cloud computing, some amount of the tasks associated with running programs are offloaded to the cloud provider. For the following, select whether it is the cloud provider or cloud user's job:

(a) Power and cooling for servers

☑ Provider / ☐ User

(b) Kernel patching/management for an infrastructure as a service cloud offering (e.g., Amazon EC2)

☐ Provider / ☑ User

(c) Maintaining/running application code for a pure infrastructure as a service offering (e.g., Amazon EC2)

☐ Provider / ☑ User

(d) Kernel patching/management for a software as a service offering (such as Google Docs)

☑ Provider / ☐ User

(e) Maintaining/running application code for a software as a service offering (such as Google Docs)

☑ Provider / ☐ User

(f) Kernel patching/management for a serverless offering (such as Amazon Lambda)

☑ Provider / ☐ User

(g) Power and cooling for servers in a serverless offering

☑ Provider / ☐ User / ☐ There are no servers, it's serverLESS!

5.2 Forms of Isolation (10 pts)

On a single shared physical host machine, applications can be run simultaneously with an illusion of running on a private address space or system by using either different processes, different containers, or different virtual machines. For the following cases, identify the isolation primitive that makes the most sense, without adding unnecessary overhead.

(a) (2 pts) Alice and Bob both want to efficiently run identical web servers that are hardwired to use network port 80.

☐ process / ☑ container / ☐ virtual machine

(b) (2 pts) Alice and Bob each want to install mutually incompatible kernel modules.

☐ process / ☐ container / ☑ virtual machine

(c) (2 pts) Alice wants to run two programs that were developed together and interact with each other frequently through interprocess communication system calls.

☑ process / ☐ container / ☐ virtual machine

(d) (2 pts) Alice wants to efficiently run two applications: one that relies Fedora's version of libc and one that relies on Ubuntu's version of libc.

☐ process / ☑ container / ☐ virtual machine

(e) (2 pts) Alice wants to package her dynamically linked application and all its dependencies so that it can efficiently be shipped to run on systems other than her own.

☐ process / ☑ container / ☐ virtual machine