CS3214 Spring 2024 Final Exam Solutions

May 6, 2024

Contents

1 Networking (24 pts) 2

1.1 Know Your Internet (10 pts) . . . 2

1.2 An HTTP Transaction (6 pts) . . . 3

1.3 Pipes vs Sockets (8 pts) . . . 3

2 Virtual Memory (13 pts) 4

2.1 On-demand Paging (9 pts) . . . 4

2.2 Overcommitting Memory (4 pts) . . . 5

3 Dynamic Memory Management (15 pts) 6

3.1 Explicit Lists (6 pts) . . . 6

3.2 Eliding Boundary Tag Footers (3 pts) . . . 7

3.3 Reconstruct this Program (6 pts) . . . 7

4 Automatic Memory Management (24 pts) 9

4.1 Fill in the Blanks (12 pts) . . . 9

4.2 Object Reachability in Javascript (6 pts) . . . 9

4.3 Verbose GC (6 pts) . . . 10

5 Virtualization (8 pts) 11

6 Essay Question: Compressed Swap (16 pts) 12

1 Networking (24 pts)

1.1 Know Your Internet (10 pts)

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

(a) Network protocols describe the format and order of messages sent and received among network entities, along with the actions to be taken on message transmission and receipt.

☑ true / ☐ false.

(b) The Internet consists of 10's of thousands autonomous systems (AS) that are interconnected.

☑ true / ☐ false.

(c) Every Internet core router maintains a forwarding database that allows it to find an address prefix matching every valid destination on the entire Internet.

☑ true / ☐ false.

(d) As the overall delay between endpoints increases, pipelined protocols must compensate by decreasing the number of packets in flight.

☐ true / ☑ false.

Pipelined protocols compensate by increasing the number of packets in flight.

(e) The higher the bandwidth of a network link, the smaller is its associated transmission delay.

☑ true / ☐ false.

(f) TCP connections are characterized by 4 parameters, which include a local address and port and a remote (or peer) address and port.

☑ true / ☐ false.

(g) Application layer protocols utilizing TCP may have to compensate for TCP's inability to preserve message boundaries.

☑ true / ☐ false.

(h) If a TCP sender retransmits a delayed packet that eventually arrived at the receiver, the TCP receiver will filter out the duplicate packet instead of delivering it.

☑ true / ☐ false.

(i) An HTTP/2 client has more incentives to create multiple transport layer connections than an HTTP/1.1 client.

☐ true / ☑ false.

Reducing the incentives for client to create multiple connections was one of the design goals of HTTP/2.

(j) HTTP servers serving different domains must be hosted on machines with different IP addresses.

☐ true / ☑ false.

No, the `Host:` header specifies the domain targeted by the request.

1.2 An HTTP Transaction (6 pts)

Consider the following HTTP transaction. A client sends the following request to your p4 server:

```
POST /api/login HTTP/1.1
Host: hazelnut.rlogin:12345
User-Agent: curl/7.76.1
Accept: */*
Content-Type: application/json
Content-Length: 44

{"username":"users24","password":"spring24"}
```

The server responds with:

```
HTTP/1.1 400 OK
Server: CS3214-Personal-Server
Add-Cookie: auth_jwt_token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAi... (elided)
Content-Type: application/json
Content-Length: 200

{"exp":1714609253,"iat":1714522853,"sub":"users24"}
```

List three errors in the HTTP response:

(a) The status code should be 200, not 400

(b) The header to set a cookie is called `Set-Cookie`

(c) The content length header is clearly wrong - the actual length on the body is far smaller - only 51 bytes.

1.3 Pipes vs Sockets (8 pts)

Compare the Unix pipes you used in p1 to the TCP sockets you used in p4. List two things that they have in common, and two things that are different.

(a) (4 pts) Examples of commonalities include

- (i) provide reliability
- (ii) provide in-order delivery
- (iii) provide a byte stream abstraction
- (iv) provide a bounded-buffer abstractions with flow control
- (v) are presented as file descriptors which includes all their conveniences, such as automatic reclamation on exit

(b) (4 pts) Examples of differences include

- (i) Pipes are unidirectional, sockets are bidirectional
- (ii) Pipes are a local IPC mechanism, sockets support communication over a network
- (iii) TCP socket require connection setup/teardown, pipes do not.

2 Virtual Memory (13 pts)

2.1 On-demand Paging (9 pts)

Consider the implementation of the `memcpy` standard library function shown below:

```
// copies n bytes from memory area src to memory area dest
void *memcpy(void *dest, const void *src, size_t n) {
    char *d = (char *)dest;
    const char *s = (const char *)src;
    while (n--) {
        *d++ = *s++;
    }
    return dest;
}
```

Assume a system that is using paged virtual memory with a global replacement policy. Let P_s be the page size. Let n be the number of bytes to be copied which is passed as the 3rd argument to `memcpy`. As necessary, express your answers to the questions below as a function of n and P_s:

(a) (3 pts) What is the minimum number of page faults that could be encountered while executing this function?

If the source and destination buffers are resident and if the code of the function itself is resident, then there are no page faults at all. Thus the minimum number of page faults is zero.

(b) (3 pts) What is the maximum number of page faults that could be encountered while executing this function? First, assume that the machine on which this program runs has plenty of physical memory and no other processes are running. Consider all segments of virtual memory.

The worst case here is that none of the buffers is resident, and also the code hasn't been paged in, but there's enough physical memory to fit both. Both source and destination array span n bytes, which requires $\lceil \frac{n}{P_s} \rceil$ pages for each. Assuming $n > 1$, these arrays thus can span a contiguous range of $\lceil \frac{n}{P_s} \rceil + 1$ virtual pages. The code of the function itself can span an additional 2 pages (if it straddles a page boundary), so the answer would be

$$2 + 2\left(\left\lceil \frac{n}{P_s} \right\rceil + 1\right)$$

(c) (3 pts) Second, answer the previous question regarding the maximum number of page faults again, but this time without assuming anything about the amount of physical memory available and with other processes possibly running.

Here, we cannot assume that both buffers (or the program text) will fit into physical memory. Even if they did, the activities of other processes may cause pages to be evicted after they have been brought in. The absolute worst case here is thrashing - the process faults on an address, the page fault handler allocates a page frame and brings in the page, but before the process has a chance to complete the memory access, another process has already stolen the page frame again. OS generally do not guarantee forward progress here due to the way page fault handling is implemented.

Thus, the answer is that there's no upper bound on the number of page faults.

(Graders note that the question did not ask for a justification of the answer.)

If you assumed that the OS will guarantee that a faulting memory access will complete after causing a page fault, the answer would be $2n + I$ where I is the overall number of machine instructions executed in the function's body. Graders should accept this answer as well.

2.2 Overcommitting Memory (4 pts)

In class, we had demoed that the Linux kernel overcommits memory, leading to scenarios where the aggregate demand exceeds the combined amount of physical memory and swap space available.

Name 2 reasons for why determining aggregate memory demand exactly is difficult:

Reasons include

(a) OS cannot know which of the allocated virtual memory will actually be accessed by a process.

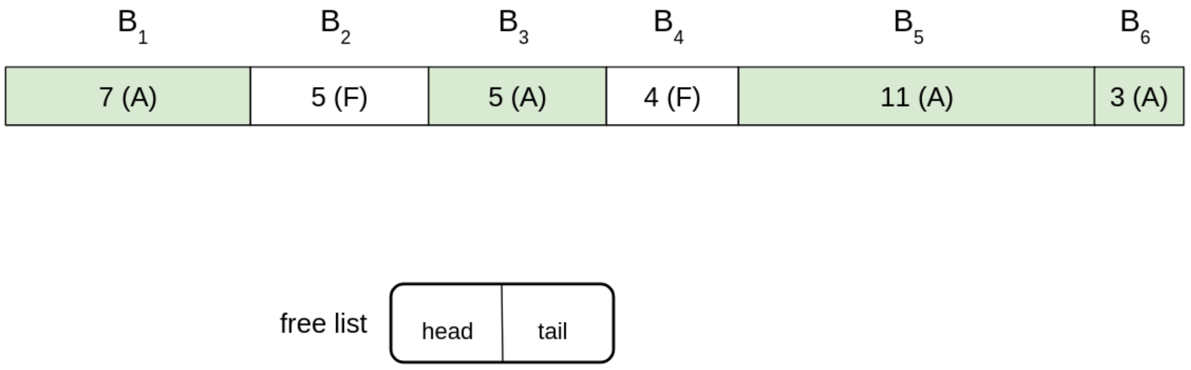
(b) Sharing memory such as when shared libraries are loaded raises an accounting problem.

(c) Predicting how much in-kernel memory as a result of processes' future interaction with the kernel is difficult, for example memory used for page tables and buffer caches.

3 Dynamic Memory Management (15 pts)

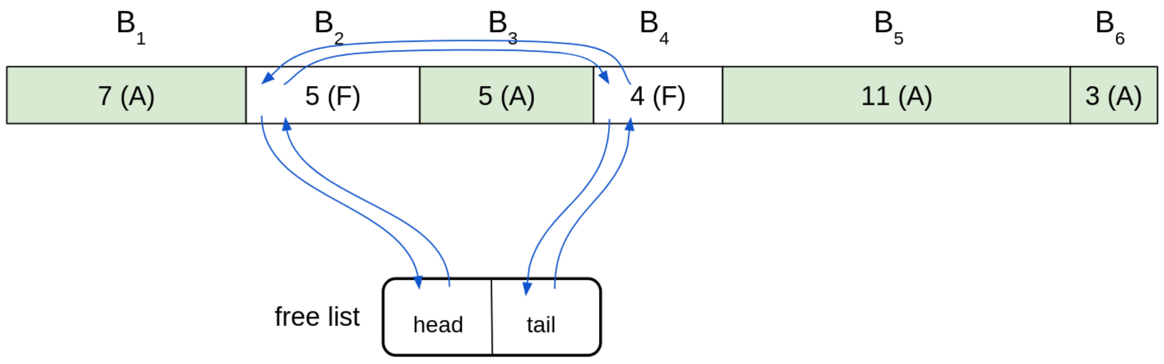
3.1 Explicit Lists (6 pts)

Consider the following sketch of a heap area managed by a dynamic memory allocator. Blocks are labeled B_1 through B_6, and each block is labeled with its size counted in words as well as whether the block is allocated (A) or free (F). The state shown is after the allocator processed a number of allocation and deallocation requests.



(a) (2 pts) Suppose this allocator uses a single explicit free list. Draw this list by adding lines and/or arrows into the sketch above. Your drawing should visualize a valid way of what this list could look like.

Your drawing should visualize a doubly-linked list connecting head, B_2, B_4, and tail. The order of B_2 and B_4 may be switched.



(b) (4 pts) Which of the following sequences of requests would fail due to external fragmentation? Ignore headers and internal fragmentation.

(i) malloc(5 words), then malloc(4 words), then free(B_6)
[✓ works] / ☐ fails.

(ii) malloc(7 words), then free(B_1), then malloc(4 words)
☐ works / [✓ fails.]

(iii) free(B_3), then malloc(9 words)
[✓ works] / ☐ fails.

(iv) free(B_5), then malloc(16 words)
☐ works / [✓ fails.]

3.2 Eliding Boundary Tag Footers (3 pts)

Boundary tags include the block size and also an in-use bit denoting whether a given block is in use (allocated) or not (free). They are ordinarily duplicated in both a header and a footer. Consider the following idea for improving your p3 allocator: add an additional bit called prev-in-use. This bit is set in a block's boundary tag header if and only if the previous block, that is, the physically left neighbor of said block is in use. Then, according to the proposal, we can eliminate the footer for all blocks that are in use. Free blocks still need a footer.

(a) (1 pt) Assuming this proposal is feasible, what performance metric would this proposal improve?

> It would improve the utilization metric or score.

(b) (2 pts) Is this a feasible proposal? Assume that the allocator performs immediate coalescing upon free(). Justify your answer.

> Yes, it would be feasible. The reason we have footers is so that we can determine the size of the left neighboring block, and using it compute where the left neighboring block starts, so that we can coalesce both blocks if both are free. However, if the left neighbor is not free, we don't need to know its size, thus we don't need a footer if we can get this information for the right neighbor's header.
>
> As a matter of fact, this is a commonly done optimization in boundary tag based allocators that some of you may have implemented in p3.

3.3 Reconstruct this Program (6 pts)

Dr. Back wrote a program called `realloc.c`. When compiled and run on our rlogin cluster, this program outputs the letter `X`.

```
$ ./realloc
X
```

When run under valgrind, however, the output is:

```
==6298== Memcheck, a memory error detector
==6298== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==6298== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==6298== Command: ./realloc
==6298==
==6298== Invalid read of size 1
==6298==    at 0x401184: main (realloc.c:8)
==6298==  Address 0x4a77054 is 20 bytes inside a block of size 2,024 free'd
==6298==    at 0x484C184: realloc (vg_replace_malloc.c:1690)
==6298==    by 0x401177: main (realloc.c:7)
==6298==  Block was alloc'd at
==6298==    at 0x484480F: malloc (vg_replace_malloc.c:442)
==6298==    by 0x401157: main (realloc.c:5)
==6298==
X
==6298==
==6298== HEAP SUMMARY:
==6298==     in use at exit: 4,048 bytes in 1 blocks
==6298==   total heap usage: 3 allocs, 2 frees, 7,096 bytes allocated
```

```
==6298== ...
==6298== For lists of detected and suppressed errors, rerun with: -s
==6298== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Based on this information, reconstruct `realloc.c` by filling out the template below:

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
    char *p =


    printf("%c\n", p[20]);
}
```

The reconstruction is shown below:

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
    char *p = malloc(2024);
    p[20] = 'X';
    realloc(p, 4048);
    printf("%c\n", p[20]);
}
```

All constants (2024, 4048, and 20) could be obtained from the valgrind output.

4 Automatic Memory Management (24 pts)

4.1 Fill in the Blanks (12 pts)

Unlike explicit (or manual) memory management schemes, automatic memory management schemes allow programmers to allocate memory, but a garbage collector is in charge of deallocation. These schemes rely on the observation that legal programs will not be able to access memory objects that are not reachable from any root. Examples of such roots include local and global variables.

When used in multi-threaded environments, some schemes require that all threads are stopped when performing their work while others allow for concurrent modifications. In the latter case, read or write barriers must be used to coordinate between the collector and mutator threads.

The widespread use of these schemes has allowed for memory-safe (or type-safe, or just safe) programming in languages such as Java, Python, or Javascript, which has helped to avoid some types of bugs that are commonly found in C or C++, such as use-after-free errors, dangling pointers, or double frees (one is enough to mention. However, this gain is paid for with increased memory consumption when performance that is comparable to malloc()-based programs is desired.

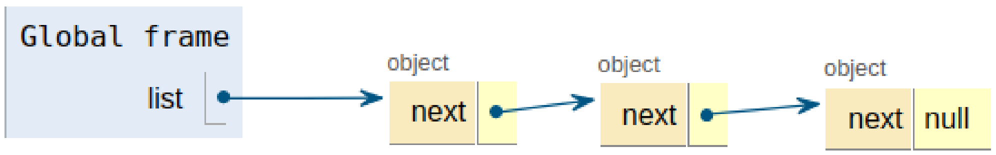
4.2 Object Reachability in Javascript (6 pts)

Consider the following JavaScript program:

```
let list = null
for (let i = 0; i < 3; i++)
   list = { next: list }
// <- consider this point
console.log("done", list)
```

Note that { `next: list` } is an object literal that creates a new object with a single field (or property) `next` that is a reference initialized with the current value of `list`.

(a) (2 pts) Sketch the live object heap as this program reaches line 4



This was basically a short version of this semester's OOM.

(b) If garbage collection happened at this point, what roots would the garbage collector need to consider?

The global variable `list` is the only root in this program.

(c) How many objects could be reclaimed via GC at this point?

Zero/None. All allocated objects are reachable from `list`.

4.3 Verbose GC (6 pts)

Shown below is the output resulting from running a Java program with the `-verbosegc` flag, which causes the JVM to output information about the garbage collection cycles it performs.

```
[0.253s][info][gc] GC(0) Pause Young (Allocation Failure) 546M->256M(1979M) 94.420ms
[0.470s][info][gc] GC(1) Pause Young (Allocation Failure) 802M->530M(1979M) 122.559ms
[0.683s][info][gc] GC(2) Pause Young (Allocation Failure) 1076M->803M(1979M) 114.388ms
[0.897s][info][gc] GC(3) Pause Young (Allocation Failure) 1349M->1077M(1979M) 114.439ms
[1.109s][info][gc] GC(4) Pause Young (Allocation Failure) 1623M->1351M(1979M) 114.311ms
[1.545s][info][gc] GC(6) Pause Full (Allocation Failure) 1625M->1625M(2171M) 175.073ms
[1.547s][info][gc] GC(5) Pause Young (Allocation Failure) 1897M->1625M(3762M) 338.826ms
[2.003s][info][gc] GC(7) Pause Young (Allocation Failure) 2595M->2111M(3762M) 261.594ms
[2.427s][info][gc] GC(8) Pause Young (Allocation Failure) 3149M->2632M(3762M) 235.987ms
[3.281s][info][gc] GC(10) Pause Full (Allocation Failure) 3152M->3152M(4190M) 346.794ms
[3.283s][info][gc] GC(9) Pause Young (Allocation Failure) 3670M->3152M(7305M) 669.138ms
[4.170s][info][gc] GC(11) Pause Young (Allocation Failure) 5038M->4098M(7305M) 507.556ms
[4.994s][info][gc] GC(12) Pause Young (Allocation Failure) 6113M->5108M(7305M) 460.425ms
[6.666s][info][gc] GC(14) Pause Full (Allocation Failure) 6119M->6119M(8134M) 684.803ms
[6.668s][info][gc] GC(13) Pause Young (Allocation Failure) 7123M->6119M(9898M) 1309.009ms
[8.398s][info][gc] GC(16) Pause Full (Allocation Failure) 8598M->7362M(9898M) 1270.819ms
```

In each line, the first number includes the time since starting the VM (e.g. 0.253s), the last number is how long the collection cycle took (e.g., 94.420ms), and the line includes information by how much it was able to reduce the amount of allocated objects (e.g., from 546MB to 256MB in the first collection, which means it was able to reclaim 290MB). The number shown in parentheses (e.g., 1979M) includes additional allocated memory, which we can ignore for this problem. The collector is a generational collector so it outputs whether the collection is a full collection or just a collection of the young generation.

Answer the following questions:

(a) (2 pts) Why do the times listed in the last column increase for each full collection cycle (those marked with `Pause Full`)?

Because the size of the live heap has increased, which leads to longer times for the mark phase of a full GC.

(b) (2 pts) What is likely to happen if this pattern continues in the way shown?

It will run out of memory.

(c) (2 pts) What can you conclude about the characteristics of the program that was run?

The program has either a memory leak or legitimately requires a very large amount of memory, for instance, because the problem size chosen was very large and/or there is an excessive amount of data structure overhead (bloat), or both.

5 Virtualization (8 pts)

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

(a) Virtual machines provide an environment in which both application and kernel code can be efficiently executed.

☑ true / ☐ false.

(b) "Deprivileging" (as a technique) refers to the idea of running a superuser container in a non-privileged virtual machine.

☐ true / ☑ false.

Deprivileging refers to running kernels that are designed to run in privileged kernel mode in a less privileged mode in order to trap and emulate any privileged instructions.

(c) Booting a container usually takes much longer than starting a virtual machine.

☐ true / ☑ false.

No, containers start up much more quickly than VMs in most cases.

(d) Both virtual machine hypervisors and container engines provide the ability to control resources such as CPU and memory.

☑ true / ☐ false.

(e) Both container images and virtual machine images benefit from copy-on-write techniques when being used by active instances (of containers and virtual machines, respectively).

☑ true / ☐ false.

(f) The efficient implementation of container engines has become possible through architectural processor extensions such as Intel's VT-x or AMD-V.

☐ true / ☑ false.

Containers do not rely on architecture support at all; virtual machines do.

(g) Container engines use virtual machines to run on operating systems that are different from the operating system for which the programs running inside the container were compiled.

☑ true / ☐ false.

(h) It possible to host a container that relies on the same or an older version of the standard libraries (such as libc) on a machine, but it is not possible to run containers that use more recent versions than what is installed on the host machine.

☐ true / ☑ false.

This poses no problem at all - I demoed running a Ubuntu 24.04 container on top a CentOS Stream system (rlogin) in class.

6 Essay Question: Compressed Swap (16 pts)

Traditionally, OS kernels page out data in physical page frames to a special area on disk called the swap space. Recently, researchers came up with the following idea: instead of sending data to swap, the data is compressed and stored in a separate area of RAM first, and sent to actual swap space on disk only when the storage space in this area has run out.

Discuss the merits and trade-offs of this idea!

Note: This question will be graded both for content/soundness of your technical arguments (10 pts) and for your ability to communicate effectively in writing (6 pts). Your answer should be well-written, organized, and clear.

This is a very open-ended question; many answers are acceptable, as long as they have some truth to them. But a complete answer deserving the full 16 points should have two parts - an advantage and a disadvantage of memory compression.

The main advantage of memory compression is to minimize how much data must be swapped out when the system is under high memory pressure. This reduces the number of costly disk or SSD accesses and improves performance. Another acceptable answers could be reducing the number of writes to storage and, therefore, improving the lifetime of storage-based SSDs. Other advantages could also be acceptable.

A disadvantage is that compressing memory data can consume a lot of CPU cycles. Another disadvantage is that compressed data are much slower to access than uncompressed data; so using some memory to store compressed data effectively reduces the amount of fast memory in the system (e.g., using 5GB out of a 10GB memory system to store compressed data means the system has only 5GB of fast memory). Again, other disadvantages are also possible.