CS 3214 Summer 2020 Final Exam Solution

August 16, 2020

Contents

Rules

- This exam is open book, open notes, and open Internet, but in a read-only way.
- You are not allowed to post or otherwise communicate with anyone else about these problems.
- You are required to cite any sources you use, except for lecture material, source code provided as part of the class materials, and the textbook.
- If you have a question about the exam, you may post it as a *private* question on Piazza. If it is of interest to others, I will make it public.
- Any errata to this exam will be published prior to 12h before the deadline.

Submission Requirements

Submit a tar file that contains the following files:

- `ObjectReachability.java` to answer Question 1.1. Add the statement asked for in part (b) to the `main()` function.
- `churn.txt` to answer Question 1.2. This will contain source code in your chosen language, but for uniformity, please give it this name.
- `dynmemory.txt` to answer Questions 2.1 and 2.2.
- `pagefault.c` to answer Question 3.2.
- `vm.txt` to answer Questions 3.1 and 3.3.
- `internet.txt` with answers to Question 4.1.
- `networking.txt` with answers to Question 4.2.

1 Automatic Memory Management (18 pts)

1.1 Object Reachability Graphs (12 pts)

(a) In systems using automatic memory management, it is important to understand how the object reachability graph changes as a result of a program's action. Figure 1 shows a snapshot of a reachability graph produced by the execution of a small Java program.

Reconstruct this program, and denote with a comment the point in time at which the reachability graph has the structure displayed in Figure 1.

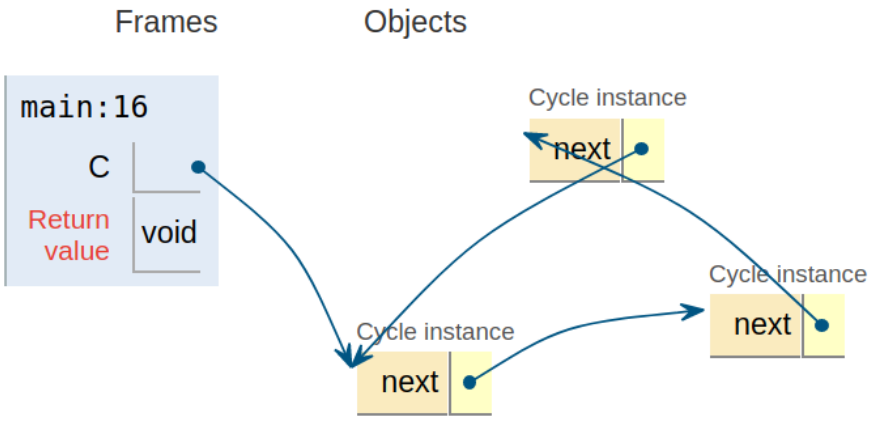


Figure 1: A snapshot of an object reachability graph produced by a Java program. On the left, roots that are part of stack frames are shown, in this case, a single static method `main` with a local variable called `C` of type `Cycle`, which contains a single field called `next`.

(b) Which Java statement, if added to the `main()` function, would turn all of the objects shown in the figure into "garbage?"

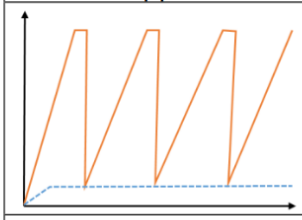
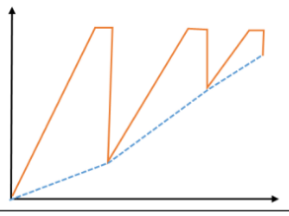
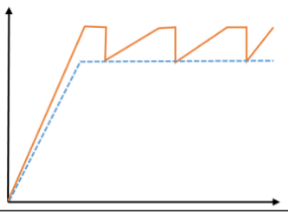
Solution

```
1  public class ObjectReachability {
2      static class Cycle {
3        Cycle next;
4        Cycle(Cycle next) {
5          this.next = next;
6        }
7        Cycle() {
8          this(null);
```

```
9         }
10      }
11
12      public static void main(String[] args) {
13        Cycle C = new Cycle();
14        C = new Cycle(C);
15        C = new Cycle(C);
16        C.next.next.next = C;
17
18        // disconnect Cycle from main stack frame (part b)
19        C = null;
20      }
21  }
```

1.2 Churn (6 pts)

An old CS3214 exam contained the following depiction of Churn, Bloat, and Leaks:

App 1	App 2	App 3
		
Check one:	Check one:	Check one:
☑ Churn	❑ Churn	❑ Churn
❑ Bloat	❑ Bloat	☑ Bloat
❑ Leak	☑ Leak	❑ Leak

- Churn is characterized by high allocation rates of garbage collectable objects: the allocation curve is steep, but live heap size is not significant.
- Bloat is characterized by high data structure overhead – the amount of live heap memory to hold an application's live data is large.
- A Leak is characterized by a steady increase in live heap size.

Write a **small** program (or function) in a garbage-collected language of your choice that fits the above definition of churn.

Solution Any program that allocates temporary objects that are not anchored to any roots will do, even as simple as

```
for (int i = 0; i < 100000; i++)
   new Integer(42);
```

2 Dynamic Memory Management (18 pts)

In project 3, you implemented a dynamic memory allocator. The performance index of your implementation reflected a weighted average of peak memory utilization and throughput. Peak memory utilization was defined as the ratio of aggregate payload size requested by the client and the amount of memory your allocator obtained from the underlying system when the aggregate payload size reached its peak. The next 2 questions assume this performance metric.

2.1 Best-Case Performance (Custom Allocators) (9 pts)

Although general purpose allocators must be designed to provide good performance for a range of workloads, for the purposes of our project, you had complete knowledge of the workload (traces) on which it would be benchmarked.

Suppose you had been asked to write an allocator that optimizes the performance index for the workload shown in the following code (and for *only* this workload):

```
#define LARGE 10000
int *p1[LARGE];
int *p2[LARGE];

void
workload()
{
    for (int i = 0; i < LARGE; i++) {
        p1[i] = malloc(240);
        p2[i] = malloc(160);
    }
    for (int i = 0; i < LARGE; i++)
        free(p2[i]);
    int * p3 = malloc(LARGE * 161);
    for (int i = 0; i < LARGE; i++)
        free(p1[i]);
    free(p3);
}
```

Describe what policies this allocator would use to maximize its performance score!

Solution. This workload contains only 3 different allocation sizes, and whatever strategy is chosen depends on only the asked-for size. Therefore, allocation can easily be implemented in constant time $O(1)$, maximizing the throughput component of the performance index. (For instance, segregated fits with exact-size or range-size policies support this.)

To maximize the utilization component, consider that the aggregate payload reaches its peak with the malloc statement on line 14. It is possible to design a strategy that asks for no more than approx. $(161 + 240) \times \text{LARGE}$ bytes from the system (ignoring overhead due to headers or alignment, but without any memory lost to internal fragmentation). For instance, blocks of memory could be placed out such that the 240 blocks are allocated before the 160 blocks like so: $240, 240, 240, \ldots, 160, 160, 160$. Then coalescing the freed blocks on line 13 can create a large block big enough to hold the allocation on line 14.

Note: though this question mainly asked for your understanding of the performance index, the example given is a bit contrived in that custom allocators generally do not have knowledge that is this precise about a workload. Usually, only common object sizes are known and the allocator optimizes for those.

Your answer needs to address both how to optimize throughput and utilization. Note that moving blocks in memory is not allowed for any malloc-style allocator used in C.

2.2 Worst-Case Performance (9 pts)

Now suppose an adversary knows your p3 allocator's policies in detail.

Design a workload that would hurt your allocator's performance index the most. It is not necessary to provide a formal proof, a description of how the workload works (using a suitable notation, possibly C code) will suffice.

If you haven't completed project 3, discuss what policies you would have chosen had you completed it, otherwise be sure to describe any applicable policies you used in your implementation.

Solution The goal of an adverse policy must be to lower throughput or utilization or both. The answer depends on what you implemented in your project. If you implemented segregated fits with a policy that looks into the next largest size class, decreasing throughput would require an attack at the largest size class, creating a large list of uncoalesced free blocks (you need to describe how to accomplish this, for instance by allocating, then freeing every other block.) If you had an explicit free list, this attack would work right away.

To diminish utilization, you could target internal or external fragmentation, or both. Targeting internal fragmentation would require allocation sizes that do not cause splitting, that is, that leave unused bytes in each allocated block.

The main weakness of all allocators, however, is that they are subject to external fragmentation since they are unable to move blocks. For any policy, it's possible to construct a workload that leaves free, but uncoalesced blocks. Simply free every other block then

allocate blocks larger than the largest such freed block. Knowing your allocator's policies, the blocks to be freed can be identified by simply running or simulating these policies since your allocator has to make a placement decision when `malloc()` is called.

3 Virtual Memory (26 pts)

3.1 Files and Memory (10 pts)

Consider the following Unix program which unfortunately lacks documentation.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/mman.h>

int
main(int ac, char *av[])
{
    if (ac != 2) {
        fprintf(stderr, "Usage: %s [fname]\n", av[0]);
        return EXIT_FAILURE;
    }

    int fd = open(av[1], O_RDONLY);
    if (fd == -1) {
        perror("open");
        return EXIT_FAILURE;
    }

    char *p = mmap(NULL, getpagesize(), PROT_READ, MAP_PRIVATE, fd, 0);
    if (p == NULL) {
        perror("mmap");
        return EXIT_FAILURE;
    }

    if (strncmp(p, "#!", 2) == 0) {
        char *c;
        int rc = sscanf(p + 2, "%ms\n", &c);
        if (rc == 1) {
```

```
            fprintf(stderr, "%s\n", c);
            return EXIT_SUCCESS;
        }
    }
    return EXIT_FAILURE;
}
```

Write a brief, man-page like description of what this program does. Be sure to include usage, a synopsis of the intended function, and a description of both the output and the exit status conventions of this utility.

Solution

`USAGE:`

`./shebang file - report if executable is a script and output its interpreter`

`DESCRIPTION:`

This program examines if a file contains a "hashbang" signature. If so, it extracts the name of the interpreter program that would be invoked to execute the script contained in the file. If the signature was found, the program outputs the name of said interpreter to its standard error stream and exits with zero, else with a non-zero exit status.

`NOTES:`

The program is implemented using the mmap(2) system call to map the file's content into its memory space.

3.2 On-demand Paging (6 pts)

Like most modern OS, Linux uses fully on-demand paged virtual memory. The following shell script uses the `time(1)` command to display the number of page faults that occur during the execution of a process:

```
#!/bin/bash
#
# Run a command and report the number of minor page faults
#
/usr/bin/time -f "%R minor pagefaults" $*
```

Testing it on an empty C program (which includes only the starter code that calls main()):

```c
// empty C program for baseline
int main() { }
```

yields an output of about 50–55 pagefaults on a contemporary Linux x86_64 machine with a 4KB page size.

Write a single-threaded C program that, when run, reports about 1050 minor page faults and meets the following conditions:

- It does not use variables of global extent (this includes global variables, global static variables, and local static variables)
- It does not use any dynamic memory (`malloc()`, `sbrk()`, `mmap()` or any system call that allocates virtual memory addresses from the OS)
- It does not use automatic local variables with a size of more than 4KB as reported by the `sizeof()` operator
- It does not use GCC's variable length arrays

In other words, the program must consume mainly memory via its stack.

Solution Any program that ensures that the stack grows and all pages on the stack are touched will work, except for the condition that no local variable be larger than 4KB. This requires either $1,000$ separate variables, or the use of recursion.

```c
#include <stdlib.h>

void recurse(int n)
{
    volatile char buf[4096] = { 0 }; // prevent compiler from optimizing this away
    buf[0]++; // cause a memory access

    if (n > 0)
        recurse(n-1);
}

// invoke as
// recurse 1000
int
main(int ac, char *av[])
{
    recurse(atoi(av[1]));
}
```

Note that the statement `buf[0]++` may not be necessary if the compiler inserts instructions to save/restore caller-saved registers on the stack, which also cause memory accesses that are roughly spaced 1 page apart. It is inserted here to prevent the compiler from removing the `buf` array altogether when compiled with optimizations.

3.3 VM Policies (10 pts)

"Virtual memory" is a set of techniques, that combines mechanisms such as address translation, MMU-based access control, page replacement, prefetching, etc. with tuned policies to achieve virtualization and protection. It was originally developed in the 1960's when main memory sizes were only a fraction of today's sizes. For instance, the departmental rlogin cluster installed in May 2020 contains 32 nodes with 384 GB of RAM each.

Select 2 different techniques underlying virtual memory and discuss specifically whether and/or how they apply to such machines with relatively large main memories. If applicable, discuss how policies used by the VM subsystem should/could be adapted for larger memory sizes. Briefly justify your arguments.

Limit: no more than one paragraph for each technique/argument.

No solution provided. We were looking for 2 independent arguments supported by (correct) facts, each focusing on a technique that is part of virtual memory. Merely citing the definition of a technique is not enough – you needed to specifically address how the importance of this technique changes (or why it does not change) with the trend to larger physical main memories. Generally aspects that relate to protection are unaffected by the main memory size. Aspects that relate to virtual address space (such as page table size in hierarchical schemes), are also not affected. Resource-related policies (such as on-demand paging or page replacement) are affected. You could argue that they're less needed, or you could argue that keeping them might allow even larger workloads.

4 Networking (38 pts)

4.1 Know Your Internet (28 pts)

Find out if the following statements related to networking are true or false. If true, just add **true**. If false, write **false** and correct the statement.

1. Link transmission delay grows with increasing bandwidth.

   False. Link transmission delay shrinks with increasing bandwidth.

2. Access networks typically provide symmetric upstream and downstream bandwidth for all subscribers.

   False. Most access networks (e.g., HFC or satellite) provide more downstream than upstream bandwidth.

3. Internet routers are required to keep track of all TCP connections whose traffic passes through them.

   False. Internet routers need not be aware of transport layer connections (though some are for traffic engineering and QoS purposes).

4. Internet routers exchange information about where packets destined for certain networks can be forwarded.

   True. (They exchange this information in routing protocols such as RIP, OSPF, or BGP)

5. TCP clients are encouraged to prefer IPv6 addresses to IPv4 addresses if a server possesses both.

   True. (This is how the planned transition to IPv6 might eventually take place.)

6. A file descriptor associated with a TCP socket will be automatically closed when the remote side closes the TCP connection.

   False. The programmer is responsible for closing the file descriptor even after this other side has closed the TCP connection or a leak will result.

7. A TCP connection will be automatically shut down when all processes that maintain file descriptors referring to the associated socket have terminated.

   True. (All file descriptors are closed on process exit and any associated resources are deallocated, including side effects such as closing a TCP connection.)

8. The 16-bit TCP port space limits a server running on a single Internet end host to an simultaneous total of no more than $65,536$ ongoing TCP connections.

   False. TCP demultiplexing uses a quadruple of (src addr, src port, dst addr, dst port), hence a server using a single port could maintain up to 2^{32+16} connections in theory.

9. TCP sockets can be used only when communicating across machines; for local connections, the programmers must use Unix pipes.

   False. TCP sockets can also be used for IPC on a machine, e.g. via 127.0.0.1

10. If a process writes data faster into a TCP connection than the remote peer is able to receive, the process is placed in the BLOCKED state until the remote peer has signaled that it is ok to continue sending data.

    True. (This TCP feature is called flow control.)

11. The HTTP protocol could be made more efficient by using numeric codes instead of header names such as `Content-Type`.

    True. (In fact, HTTP/2.0 Header Compression (HPACK, RFC 7541) does this.)

12. An HTTP user agent such as browser decides whether to include a cookie in an outgoing HTTP request primarily based on the targeted URL.

    True. (Cookies are maintained separately on a per-domain + path basis.)

13. If JWT tokens are leaked, they can be used by anyone to impersonate a user until they expire.

    True. (A server will accept a valid token no matter who its bearer is.)

14. A user in possession of a server's secret key is able to extend the "expires at" claim in a JWT token they have previously obtained from that server.

    True. (Yes, possession of the secret key allows anyone to manufacture valid tokens.)

4.2 Mixing Threads and Sockets (10 pts)

Consider the following program, which its programmer unfortunately failed to document:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netdb.h>
#include <pthread.h>

static int
dial(char *host, char *port)
{
    struct addrinfo hint = {
        .ai_flags = AI_CANONNAME | AI_NUMERICSERV | AI_ADDRCONFIG,
        .ai_protocol = IPPROTO_TCP
    };

    struct addrinfo *info;
    int rc = getaddrinfo(host, port, &hint, &info);
    if (rc != 0)
        gai_strerror(rc), exit(EXIT_FAILURE);

    while (info) {
        int s = socket(info->ai_family,
                       info->ai_socktype,
```

```
                        info->ai_protocol);
        if (s < 0)
            perror("socket"), exit(EXIT_FAILURE);

        if (connect(s, info->ai_addr, info->ai_addrlen) == 0)
            return s;
        close(s);
    }
    exit(EXIT_FAILURE);
}

struct fdpair {
    int from, to;
};

static void *
shovel(void *_data)
{
    struct fdpair * c = _data;
    char buf[2048];
    int bread;
    while ((bread = read(c->from, buf, sizeof buf)) > 0)
        write(c->to, buf, bread);
    return NULL;
}

int
main(int ac, char *av[])
{
    int s = dial(av[1], av[2]);
    struct fdpair p1 = { .from = s, .to = STDOUT_FILENO };
    struct fdpair p2 = { .from = STDIN_FILENO, .to = s };
    pthread_t t[2];
    pthread_create(t, NULL, shovel, &p1);
    pthread_create(t, NULL, shovel, &p2);
    for (int i = 0; i < 2; i++)
        pthread_join(t[i], NULL);
}
```

Write a brief, man-page like description of what this program does. Be sure to include usage and a synopsis of its intended function. Denote any limitations you may spot as well.

Solution

`USAGE:`

`./nc host port - connect standard in/out stream to a remote network server`

`DESCRIPTION:http://get.webgl.org/`

This program accepts two arguments, *hostname* and *port* number, representing the DNS name or IP address of an Internet host, and a numeric port number. It will attempt to resolve the DNS name based on the host configuration and connectivity to find a IPv6 and/or IPv4 address, and then attempts to connect to these address in order. Once a connection is obtained, the program copies data from its standard input stream to the connection, and data received from the TCP connection to its standard output stream, using 2 threads.

The program terminates with exit status 0 when both connections are closed or encounter an error condition. If the program can't connect, it exits with a non-zero exit status.

`BUGS:`

A shortcoming of this implementation is that the program needs to be terminated by the user, for instance with SIGINT if the network connection is not closed by the other side or the user doesn't type Ctrl-D to end the program's standard input stream (unless redirected).

Because of this shortcoming, an actual bug went unnoticed in the second call to `pthread_create` which overwrote the `pthread_t` id returned from the first call. The correct version is shown below.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <errno.h>
5  #include <unistd.h>
6  #include <sys/socket.h>
7  #include <netdb.h>
8  #include <pthread.h>
9
10 static int
11 dial(char *host, char *port)
12 {
13     struct addrinfo hint = {
```

```
        .ai_flags = AI_CANONNAME | AI_NUMERICSERV | AI_ADDRCONFIG,
        .ai_protocol = IPPROTO_TCP
    };

    struct addrinfo *info;
    int rc = getaddrinfo(host, port, &hint, &info);
    if (rc != 0)
        gai_strerror(rc), exit(EXIT_FAILURE);

    while (info) {
        int s = socket(info->ai_family,
                       info->ai_socktype,
                       info->ai_protocol);
        if (s < 0)
            perror("socket"), exit(EXIT_FAILURE);

        if (connect(s, info->ai_addr, info->ai_addrlen) == 0)
            return s;
        close(s);
    }
    exit(EXIT_FAILURE);
}

struct fdpair {
    int from, to;
};

static void *
shovel(void *_data)
{
    struct fdpair * c = _data;
    char buf[2048];
    int bread;
    while ((bread = read(c->from, buf, sizeof buf)) > 0)
        write(c->to, buf, bread);
    return NULL;
}

int
main(int ac, char *av[])
{
```

```
55      int s = dial(av[1], av[2]);
56      struct fdpair p1 = { .from = s, .to = STDOUT_FILENO };
57      struct fdpair p2 = { .from = STDIN_FILENO, .to = s };
58      pthread_t t[2];
59      pthread_create(t, NULL, shovel, &p1);
60      pthread_create(t+1, NULL, shovel, &p2);
61      for (int i = 0; i < 2; i++)
62          pthread_join(t[i], NULL);
63  }
```

The functionality is mini-version of the popular netcat (nc) program, e.g. here is an example session talking to google.com:

```
$ ./mysterynetworktool www.google.com 80
HEAD / HTTP/1.0
Host: www.google.com

HTTP/1.0 200 OK
Content-Type: text/html; charset=ISO-8859-1
P3P: CP="This is not a P3P policy! See g.co/p3phelp for more info."
Date: Sat, 15 Aug 2020 02:26:36 GMT
Server: gws
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
Expires: Sat, 15 Aug 2020 02:26:36 GMT
Cache-Control: private
Set-Cookie: 1P_JAR=2020-08-15-02; expires= ...
Set-Cookie: NID=204=hUqKXxfb5yqpqp0 ...
domain=.google.com; HttpOnly
```