CS 3214 Fall 2021 Final Exam Solution

December 13, 2021

Contents

1 Networking (28 pts) **3**
- 1.1 Know Your Internet (10 pts) 3
- 1.2 Single Transaction HTTPS Client (18 pts) 4

2 Virtual Memory (32 pts) **9**
- 2.1 Understanding Page Faults (8 pts) 9
- 2.2 Files and Memory (12 pts) 12
- 2.3 `malloc()` and virtual memory (12 pts) 14

3 Virtualization and Containers (10 pts) **17**

4 Automatic Memory Management (20 pts) **18**
- 4.1 Runaway OOM (6 pts) 18
- 4.2 Object Reachability Graphs (14 pts) 20

5 More Memory, More Problems (10 pts) **22**

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 Discourse, shared with the instructors only. If it is of interest to others, we will make it public.
- Any errata to this exam will be published prior to 12h before the deadline. Please check all public posts with tag `final` on Discourse.

1 Networking (28 pts)

1.1 Know Your Internet (10 pts)

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

1. To ensure that a network is reachable from anywhere on the Internet, a network operator must publish its IP addresses to the neighboring networks to which it is connected, who in turn relay this information to their neighbors, and so on.

   [**True**]

2. Internet Service Providers typically build (or lease or buy) fiber optical links to the rest of the Internet whose aggregate bandwidth is far less than what their access networks to their customers could support.

   [**True**]

3. If packets travel to their destination on the Internet mostly over fast fiber optic network links then their propagation delay becomes negligible.

   [**False**] The bandwidth of the link (how fast it is) doesn't matter, propagation delay is determined by the speed of light (roughly $0.6c$ for fiber) which over large geographical distances and multiple hops is no longer negligible.

4. When two global IPv4 or IPv6 addresses do not share any prefix in their bitwise representation, then the network interfaces to which they are assigned must be located in different networks.

   [**True**]

5. Because 32-bit IPv4 addresses form a subset of the 128-bit IPv6 address space, IPv6 hosts can send IP packets to IPv4 hosts, but not the other way around.

   [**False**] IPv4 and IPv6 form entirely separate networks with entirely separate address spaces. IPv4 addresses are not a subset of IPv6 addresses.

6. A rogue Internet Service Provider can inject custom JavaScript into the web sites their customers are visiting unless the customer visits only secure (https://) websites.

   [**True**]

7. To establish a TCP connection to a server, a client program will need to know both at least one IP address of the machine on which the server runs and a port number.

   [**True**]

8. Network Address Translation (NAT) devices that perform address translation between private and public networks can rewrite a forwarded packet's IP address, but they cannot rewrite the port number contained in the packet.

   [False] In fact they must rewrite the port number also since multiple internal hosts may independently choose the same port number, but the outgoing port number must be unique to identify the connection so that response packets can be properly forwarded.

9. Modern transport protocols such as QUIC are implemented as a layer on top of TCP in order to compensate for some of its shortcomings.

   [False] QUIC is implemented on top of UDP, it provides an application-specific alternative to TCP.

10. When the transition to HTTP/3 is complete, web developers will have to port their client-side applications to avoid having them break.

    [False] The update to HTTP/3 will be transparent to web applications, no updates are necessary (though some applications may be enhanced to use some of the new features).

1.2 Single Transaction HTTPS Client (18 pts)

During project 4, you learned how to extend an HTTP/1.0 server implementation to handle persistent HTTP/1.1 connections and support multiple clients. All the focus was on the server side of an HTTP implementation. This question focuses on the client side.

To make it more fun, let's implement a mini-http client for a single transaction. Unfortunately for this assignment, but fortunately for the greater good of society, there are only few servers left on the Internet that support HTTP – most will accept only HTTPS connections or redirect to a https:// URL.

For instance, if you visit the National Weather Service's API at http://api.weather.gov/ it will respond with a 301 status code asking you to visit https://api.weather.gov/ instead.

Fortunately for us, we can use the power of Unix's system call API and our knowledge from project 1 to link an HTTP client to a program that can communicate with HTTPS servers. One such program is called `openssl`.

Complete the provided program `httpclient.c` minimally so that it can perform a single HTTP transaction with the NWS weather service. `httpclient` will be invoked with 2 arguments: the domain name of the HTTPS server hosting the API, and the path component of the API endpoint. For instance, for https://api.weather.gov/gridpoints/RNK/56,64/forecast the domain name would be `api.weather.gov` and the path would be `/gridpoints/RNK/56,64/forecast`.

The HTTP client program should output the body of the response to its standard output stream so that it can be invoked as shown below:

```
$ ./httpclient api.weather.gov /gridpoints/RNK/56,64/forecast \
    | jq '.properties.periods[0].name, .properties.periods[0].detailedForecast'
"Tonight"
"Partly cloudy, with a low around 31. Southwest wind around 5 mph."
```

Rules/Hints:

- You must complete the provided program.
- For the communication between your program and openssl you should use a Unix domain socket pair (the code for this is given to you). A socket pair works much like the pipes you are familiar with, except that (unlike for pipes) each end refers to a bidirectional connection. Data sent to the first socket is received by the second, and data sent to the second is received by the first – by contrast, for pipes, there was a dedicated read and a dedicated write end.
- The openssl program, when invoked in the way shown in the code, will establish a secure connection to the target, and then forward anything it reads from its standard input to the target. Anything received from the target will in turn be output to its standard output stream. For example, `openssl s_client -connect google.com:443 -quiet -verify_quiet` would connect to Google.com. You may assume that `openssl` is in your path.
- You may assume that the server to which you talk provides a conforming implementation of HTTP/1.1, as such, handling errors that result from a misbehaving server is not required.
- You may assume that the size of the body does not exceed 64KB.
- You may not use functions banned in CS3214.
- You may use any function provided in the GNU C environment on rlogin. Hint: a sufficiently working implementation can be created using only the functions: close, dup2, execvp, snprintf, write, read, and strstr.
- You may not use the `exec()` family of calls or `system(3)` to invoke any other system utility except for `openssl`.
- You must handle "short" reads - do not assume that you can consume a response in a single `read(2)` call.
- The NWS Weather API requires that you specify an user agent (see [URL]). Provide an appropriate HTTP header to that end. The name of the user agent does not matter.

- For the purposes of this program, error handling is not required, but you may find it useful nevertheless. However, under normal operation, do not output anything else to standard output.
- The server that hosts the NWS API service is reachable through a reverse proxy server provided by a CDN provider. This provider likely hosts many domains at fewer IP addresses and you must send an appropriate header to direct the reverse proxy server to the proper domain.
- Since you only need to support a single transaction, think about a way to encourage the server to close the connection after sending its response so that you do not have to rely on the server's content length header. openssl will exit once the server closes the connection, which will close all file descriptors it has open, which you can recognize in the customary way. Your program should not hang because the server is keeping the connection open for more requests.
- Your implementation should not hardwire the domain or path, but we will test it only with https://api.weather.gov/gridpoints/RNK/56,64/forecast in the way shown above.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <stdbool.h>
#include <sys/socket.h>

int
main(int ac, char *av[])
{
    char *domain = av[1];
    char *path = av[2];

    int sockets[2];
    int rc = socketpair(PF_LOCAL, SOCK_STREAM, 0, sockets);
    if (rc != 0) {
        perror("socketpair");
        return EXIT_FAILURE;
    }

    if (fork() == 0) {
```

```
        char buf[1024];
        snprintf(buf, sizeof buf, "%s:443", domain);
        char *argv[] = { "openssl", "s_client", "-connect",
                            buf, "-quiet", "-verify_quiet", NULL };
        close(sockets[0]);

        /* complete this block so that sockets[1] appears on both
         * standard input and standard output of this process, then
         * execute openssl with the arguments shown above */
        return EXIT_FAILURE;
    }

    close(sockets[1]);

    // at this point, file descriptor sockets[0] refers to a TLS connection
    // to the target HTTPS server.  Implement a single HTTP transaction minimally
    // and output the body of the response to standard out.
    // you may assume that the response is less than 64KB
    const int MAX_RESPONSE_SIZE = 65536;

    // complete this
}
```

A possible minimal solution requires sending a `GET` request with `Host:` and `User-Agent:` header. Use either HTTP/1.0 or include a `Connection: close` header when using HTTP/1.1 to get the server to close the connection, then read the response until EOF. To find the body, scan for the first CRLFCRLF sequence.

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

int
main(int ac, char *av[])
{
    char *domain = av[1];
    char *path = av[2];
```

```
int sockets[2];
int rc = socketpair(PF_LOCAL, SOCK_STREAM, 0, sockets);
if (rc != 0) {
    perror("socketpair");
    return EXIT_FAILURE;
}

if (fork() == 0) {
    char buf[1024];
    snprintf(buf, sizeof buf, "%s:443", domain);
    char * argv [] = { "openssl", "s_client", "-connect",
                       buf, "-quiet", "-verify_quiet", NULL };
    close(sockets[0]);
    dup2(sockets[1], STDOUT_FILENO);
    dup2(sockets[1], STDIN_FILENO);
    close(sockets[1]);
    execvp(argv[0], argv);
    perror("execvp");
    return EXIT_FAILURE;
}

close(sockets[1]);

// use of HTTP/1.0 will result in an automatic close by the other end
// alternatively, use HTTP/1.1 and send a `Connection: close` header
char buf[1024];
snprintf(buf, sizeof buf, "GET %s HTTP/1.0\r\n"
        "User-Agent: CS3214 Weatherman\r\n"
        "Host: %s\r\n"
        "\r\n", path, domain);
// send request
write(sockets[0], buf, strlen(buf));

// slurp entire response into buffer
const int MAX_RESPONSE_SIZE = 65536;
char resp[MAX_RESPONSE_SIZE];
char *pos = resp;
ssize_t r;
while (pos - resp <= MAX_RESPONSE_SIZE &&
      (r = read(sockets[0], pos,
```

```
                        MAX_RESPONSE_SIZE - (pos - resp))) > 0) {
            pos += r;
        }
        // header ends with an empty line
        // find first byte after header, that's where the body starts
        char *body = strstr(resp, "\r\n\r\n") + 4;
        write(STDOUT_FILENO, body, pos - body);
}
```

2 Virtual Memory (32 pts)

2.1 Understanding Page Faults (8 pts)

When executed on a Linux system that uses on-demand paged memory, the following program causes 769 minor pagefaults (on x86_64 with 4KB pages). List the line numbers on which these page faults occur, along with how many page faults occur on each line you list.

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include <sys/mman.h>
4  #define M 256
5  #define S 4096
6
7  void *f1()
8  {
9      char *m = malloc(M * S);
10     return m;
11 }
12
13 void *f2()
14 {
15     char *m = malloc(M * S);
16     for (int i = 0; i < M; i++)
17         m[i*S] = 1;
18     return m;
19 }
20
21 void *f3()
22 {
23     char *m = malloc(M * S);
```

```
24      for (int i = 0; i < S; i++)
25          m[i] = 1;
26      return m;
27  }
28
29  char bigglobal[M*S];
30
31  void k()
32  {
33      void * base = mmap(NULL, M*S, PROT_READ | PROT_WRITE,
34                                    MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
35      memcpy(base, bigglobal, M*S);
36  }
37
38  int
39  main()
40  {
41      f1();
42      f2();
43      f3();
44
45      memset(bigglobal, 0xff, M*S);
46      k();
47  }
```

Note: if you actually run this program you would observe additional page faults that may happen as "background noise," perhaps as a result of updating internal C library data structures. Ignore those in this question and focus only on the page faults directly attributable to the code presented. Assume that the program is compiled as is without any optimizations.

[Solution] The number of pagefaults is shown in the following table and also annotated below.

Line	Number
17	256
25	1
35	256
45	256

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include <sys/mman.h>
4  #define M 256
```

```
5   #define S 4096
6
7   void *f1()
8   {
9       char *m = malloc(M * S);
10      return m;
11  }
12
13  void *f2()
14  {
15      char *m = malloc(M * S);
16      for (int i = 0; i < M; i++)
17          m[i*S] = 1;       // 256 faults
18      return m;
19  }
20
21  void *f3()
22  {
23      char *m = malloc(M * S);
24      for (int i = 0; i < S; i++)
25          m[i] = 1;         // 1 fault
26      return m;
27  }
28
29  char bigglobal[M*S];
30
31  void k()
32  {
33      void * base = mmap(NULL, M*S, PROT_READ | PROT_WRITE,
34                                    MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
35      memcpy(base, bigglobal, M*S);    // 256 faults
36  }
37
38  int
39  main()
40  {
41      f1();
42      f2();
43      f3();
44
45      memset(bigglobal, 0xff, M*S);   // 256 faults
```

```
46      k();
47  }
```

The malloc() or mmap calls do not produce pagefaults (other than noise when the memory allocator updates header data), the pagefaults occur when accessing the data. Function f2 accesses all 256 pages allocated, function f3 only the first page. The memset() operation is the first access to the global variable bigglobal, which causes pagefaults, as does accessing the mmap'ed anonymous memory on line 35. Total number is $3 \times 256 + 1 = 769$ as given in the problem.

2.2 Files and Memory (12 pts)

In a previous CS 3214 exam, the following program was presented. This program could be used to pseudo-randomly corrupt executables:

```
// run as
// ./randomize_it <old> <new> <seed>
// e.g.
// ./randomize_it cush cushr 42
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <fcntl.h>

int
main(int ac, char *av[])
{
    int from = open(av[1], O_RDONLY);
    int to = open(av[2], O_CREAT | O_TRUNC | O_WRONLY, 0777);
    int seed = atoi(av[3]);
    srandom(seed);
    char buf[1024]; // change one byte every 1024
    size_t bread;
    while ((bread = read(from, buf, sizeof buf)) > 0) {
        buf[random() % bread] = random();
        write(to, buf, bread);
    }
}
```

Change this program to use `mmap()` instead of `read()` and `write()`.

Unlike the given version, your new version should corrupt the executable in place (rather than making copy of it). It should take two arguments `filename` and `seed` instead of three.

In other words, the effect of running

```
# ./randomize_it cush cushr 42
```

on the given program should be identical to the effect of running

```
# cp cush cushr
# ./randomize_it_mmap cushr 42
```

where `randomize_it_mmap` is your program.

[Solution]

```c
// run as
// ./randomize_it_mmap <file> <seed>
// e.g.
// ./randomize_it_mmap cushr 42
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <fcntl.h>
#include <assert.h>
#include <sys/stat.h>
#include <sys/mman.h>

int
main(int ac, char *av[])
{
    int fd = open(av[1], O_RDWR);
    int seed = atoi(av[2]);
    srandom(seed);

    struct stat st;
    fstat(fd, &st);
    char *addr = mmap(NULL, st.st_size, PROT_WRITE, MAP_SHARED, fd, 0);
    assert (addr != MAP_FAILED);

    // randomize one byte every 1024, taking care of the last
    // chunk which may be smaller than 1024
    for (size_t i = 0; i < st.st_size; i += 1024) {
        size_t chunksize = st.st_size - i;
        if (chunksize > 1024) chunksize = 1024;
        int rhs = random();
        addr[i + random() % chunksize] = rhs;
```

```
    }
}
```

Note: the original program was flawed in that it included implementation-defined behavior, which was irrelevant for the purposes of producing a randomly corrupted executable, but which prevented you from creating a program that behaves identically. Specifically, C does not make any guarantees as to whether the call to `random()` on the left hand side or the call to `random()` on the right hand side would occur first. These calls have side effects, however, in that the advance the random number generator's sequence.

As a result, we accepted solutions that either ignored the issue in the way the provided code did, or that forced an evaluation in either order. (The sample solution calls `random()` to first obtain a random number for the right hand side.) When compiled as in the original program, the current version of gcc appears to produce code that evaluates the right-hand side first. In all 3 cases, the behavior of your program should have matched of either

```
buf[random() % bread] = random();
```

when compiled with gcc on the rlogin machines, or

```
int rhs = random();
buf[random() % bread] = rhs;
```

or

```
int lhs = random() % bread;
buf[lhs] = random();
```

2.3 `malloc()` and virtual memory (12 pts)

Similar (but not quite identical) to how your user-level memory allocator in project 3 requested memory in the form of virtual address ranges via `mem_sbrk()`, Linux's `malloc()` implementation requests virtual address ranges from the OS. The kernel provides an interface via `/proc/self/statm` to query about a process's total virtual memory consumption.

We wrote the following malloc test program:

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

/* This function uses Linux's /proc interface to inquire with
 * the kernel about how much virtual memory this process
 * uses. */
```

```
static long
get_virtual_memory_size()
{
    long pages;
    FILE *f = fopen("/proc/self/statm", "r");
    fscanf(f, "%ld", &pages);   // unit is pages
    fclose(f);
    return pages * sysconf(_SC_PAGESIZE);
}

int
main(int ac, char *av[])
{
    if (ac < 3) {
        printf("Usage: %s n m\n", av[0]);
        abort();
    }
    long n = atol(av[1]), m = atol(av[2]);

    printf("allocating %ld x %ld bytes, total = %ld\n", n, m, n * m);

    char **p = calloc(n, sizeof(char*));
    for (int i = 0; i < n; i++)
        p[i] = malloc(m);

    printf("after malloc this process consumes %ld KB of virtual memory\n",
            get_virtual_memory_size()/1024);

    for (int i = 0; i < n; i++)
        free(p[i]);

    printf("after free this process consumes %ld KB of virtual memory\n",
            get_virtual_memory_size()/1024);
}
```

When invoked with two numbers n and m this program will perform n allocations of m bytes, then measure the process's virtual memory consumption. It will then free those allocations and measure again.

In Scenario 1, we perform $500,000$ allocations of $1,000$ bytes:

```
$ ./malloc 500000 1000
allocating 500000 x 1000 bytes, total = 500000000
```

```
after malloc this process consumes 500372 KB of virtual memory
after free this process consumes 500372 KB of virtual memory
```

In Scenario 2, we perform $1,000$ allocations of $500,000$ bytes:

```
$ ./malloc 1000 500000
allocating 1000 x 500000 bytes, total = 500000000
after malloc this process consumes 496368 KB of virtual memory
after free this process consumes 4368 KB of virtual memory
```

1. (6 pts) Investigate why these scenarios result in such dissimilar behavior, despite the overall amount of memory requested being similar.

   [Solution] Using strace you can find that the many/small case results in an expansion of the heap using the sbrk() call, similar to how your p3 allocator worked. The heap is grown with each call, but the free() calls do not result in shrinking the heap.

   In the few/large case, the malloc() implementation uses mmap(). Thus, a free() call will munmap(), giving the virtual address space back to the OS.

2. (4 pts) Interpret and explain these results from the perspective of the designer of Linux's malloc()/free() implementation!

   [Solution] The issue of when an allocator should return memory to the OS is a tricky one. On the one hand, returning memory to the OS means that the OS is no longer responsible for either setting aside physical memory for this data or maintaining a copy of the data in swap (provided the application touched the virtual addresses in question). This frees up physical memory that other processes can then use.

   On the other hand, subsequent malloc() requests may reuse this very same memory, something the memory allocator cannot know if it will happen, but typically assumes that it will. If it does, then the memory would need to be rerequested from the OS, which is more expensive than reusing already obtained memory. In addition, checking whether memory can be returned to the OS comes at a cost that would need to be paid in each free() call. That's why GNU's malloc has the observed behavior, see Bug 27103 - glibc fails to free memory for more discussion.

   On the other hand, for a larger (and presumably less frequent) allocation, the allocator chooses to use mmap() which upon unmapping returns the memory automatically. The use of mmap() has the advantage that the OS can find a space in the virtual address space that's separate from the contiguous heap to which sbrk() refers.

3. (2 pts) Describe a workload that may be adversely affected by the implementation choices made by Linux's memory allocator, as evidenced by the behavior for this test.

   [Solution] Workloads that have allocation patterns where large areas of the heap are allocated and freed and then not reused would benefit from a more aggressive strategy of trimming virtual memory that the allocator hasn't given out to its client.

An example is Ruby, see here for more information. In my opinion, simply shrinking the heap more aggressively is unlikely to result in significant performance improvements since much of the unused memory appears to be inside the heap.

Guidance for graders. In part 1, we're looking for an understanding that the heap is grown but not easily shrunk, whereas mmap/munmap results in immediate reclamation. For part 2, we're looking for the insight that in general retaining memory in anticipation of future use is a good strategy and that not doing so incurs an expense, even though there are benefits of not retaining it. For part 3, the key is that workloads may produce allocation patterns that result in low utilization of memory.

3 Virtualization and Containers (10 pts)

Find out if the following statements related to virtual machines and/or containers are true or false.

If true, just write **true**. If false, write **false** and provide the corrected statement.

1. Both virtual machines and containers use direct execution for the user space processes contained in them.

   [True]

2. Both virtual machines and containers provide the ability to control the amount of resources (such as CPU or memory) used by the applications running inside them.

   [True]

3. To build a container image you must first compile a kernel for the version of Linux you chose for the container image.

   [False] Container engines use the kernel of the host to execute processes inside a container.

4. If a virtual machine monitor perfectly emulates the environment provided by real hardware, then an off-the-shelf OS such as Windows or Linux supporting said hardware does not require any adaptation to be able to run on top of this monitor.

   [True] (The guest OS is unaware of the fact that it is running virtualized. This is not withstanding the fact that guest OS can be enlightened, or paravirtualized, to run better inside virtualized environments.)

5. When a machine's OS kernel crashes due to a bug, any containers running on the same machine can be migrated to another machine.

   [False] Since container engines use the host kernel, a host kernel crash will render them immediately inoperable, preventing any migration.

6. Container engines such as Docker rely on hardware features that have become widely available only on recent microprocessor architectures.

   [False] Containers are entirely software constructs and unlike virtual machines do not require hardware support. (This is despite the fact that some container engines can benefit from hardware support, for instance, recent versions of Docker run better when Hyper-V is enabled on Windows which in turn requires recent hardware features, although it is possible to run Docker without this.)

7. Programs always have to be recompiled to run inside a container or virtual machine to ensure compatibility with the instruction set architecture (ISA).

   [False]. Both containers and virtual machines use the host machine's ISA. A statically compiled executable will run inside or outside a container or virtual machine without needing recompilation. Dynamically linked programs may require recompilation, but not for the purposes of ensuring ISA compatibility (rather to ensure that shared libraries can be found inside the container).

8. A user can be an administrative user inside a container even if they do not have administrative privileges on the OS hosting the container.

   [True] (You saw this in ex5 where you could be root in your container.)

9. The execution of container images written for one OS on another OS either requires a virtual machine or a system call emulation layer.

   [True] (This is how Docker for Windows can run Linux/OCI containers.)

10. Container runtimes extensively exploit Copy-on-Write (CoW) techniques to manage their file systems based on the container image, but virtual machine monitors cannot use these techniques for the virtual disks they provide to their guests.

    [False] Virtual machine hypervisors can use the same or similar CoW techniques for their disks. See Qcow for QEMU for an example.

4 Automatic Memory Management (20 pts)

4.1 Runaway OOM (6 pts)

During ex4, an unnamed student tried to reconstruct the OOM.java program that was supposed to cause the JVM to run out of memory, but ended up inadvertently starting a JVM process that ran for 13 days, consuming over $21,000$ CPU hours in the process. A snapshot of htop is shown in Figure 1.

For obvious reasons, we will not ask you to reconstruct their program. Please do not attempt this. Instead, we ask this:

Figure 1: A snapshot of htop on the node in question

Did this student (inadvertently) create an example of a Leak, Churn, or Bloat? Justify your answer!

[Solution]

This was an example of churn. If it had been a leak or bloat, the size of the live memory would have increased beyond the 64MB heap limit, which would have resulted in the JVM to stop and produce a heap dump. The htop output showed that the JVM was run with the `-Xmx64m` flag.

Instead, the JVM continuously allocated objects which quickly became garbage, garbage collected them, and repeated this process ad infinitum (or more precisely, until this process was killed). To make matters worse, this code also spawned additional short-lived threads whose allocation and deallocation added even more churn.

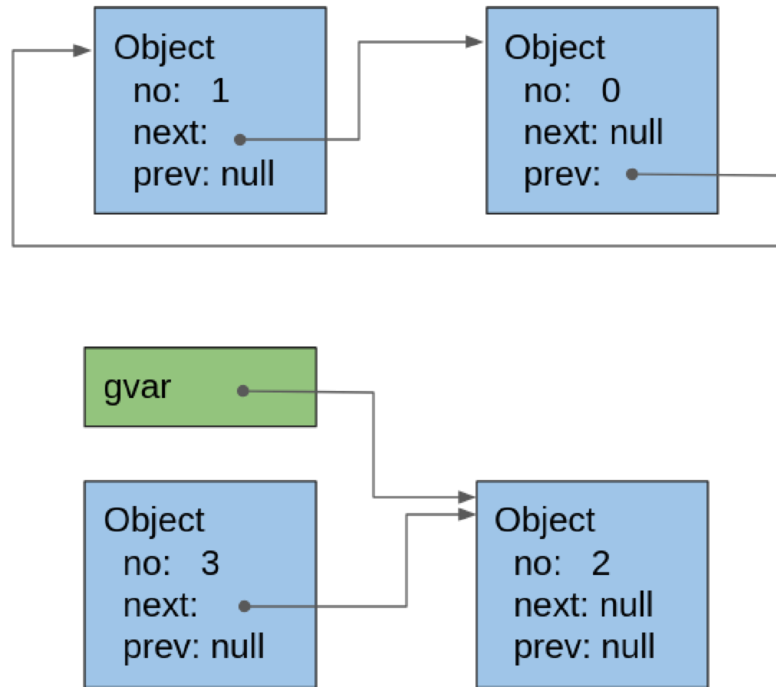
4.2 Object Reachability Graphs (14 pts)

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. Consider the following Java program:

```
1  public class A {
2    static Object gvar;
3    static class Object {
4      static int _cnt;
5      int no;
6      Object next;
7      Object prev;
8      Object(Object next) {
9        this.no = _cnt++;
10       this.next = next;
11     }
12   }
13   public static void main(String []av) {
14     Object b = new Object(null);
15     Object a = new Object(b);
16     b.prev = a;
17     // Point X
18     a = b = null;
19     gvar = new Object(new Object(null));
20     gvar = gvar.next;
21     // Point Y
22   }
23 }
```

1. Draw a sketch of the heap as it would appear on line 21 (marked as Point Y). Assume that no garbage collection has taken place. Clearly denote inter-object relationships and values of object fields. You may use a drawing program or pen and paper and take a photo or scan.

   [Solution]



   Your sketch should show 4 objects with their `no` field being 0, 1, 2, and 3. Objects 0 and 1 form a cycle where 1's next field points to 0 and 0's prev field points to 1. Object no 3's next field points to Object no 2. The global variable `gvar` is a root that points to Object no 2 as well.

   (Additional elements, such as the nulled-out variables a and b in main's stack frame, or `av`, could be shown but were not necessary since they are not connected to the heap or not known.)

2. If, hypothetically, a garbage collection took place on line 17 (marked as Point X), which roots would the garbage collector need to traverse, and what would those roots be referring to?

   [Solution] On line 17, the roots to be traversed are the local variables `a` and `b` in main's stack frame.

It's also correct (but for the purposes of this question not required) to mention `gvar` (which is still `null` on line 17, so no traversal would be necessary) and possibly `av` whose value is not apparent from the program.

5 More Memory, More Problems (10 pts)

You are designing a new extreme-scale computer for emerging Deep Learning applications that require over 1000 TB of RAM. You are considering design choices such as operating system, hardware, and power supply. Very briefly, i.e., in a couple of sentences max, list two main challenges that systems designers would face in supporting such large memory.