CS 3214 Fall 2020 Final Exam Solutions

December 17, 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, we 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:

- `networking.txt` with answers to Questions 1.1 and 1.2.
- `A.java` to answer Question 2.1.
- `garbagecycle.txt` to answer Question 2.2. This will contain source code in your chosen language, but for uniformity, please give it this name.
- `leakanalysis.txt` to answer Question 2.3.
- `vm.txt` to answer Questions 3.1 and 3.2.
- `protection.txt` with answers to Question 4.
- `allocator.txt` with answers to Question 5.

Important: Please download your tar file after you have submitted it and check the content to ensure you did not upload a corrupted tar file.

1 Networking (28 pts)

1.1 Know Your Internet (14 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. In IP networks that are designed with redundant links, traffic will automatically be rerouted should one of the links fail.

   True. Routing protocols are used to find feasible routes even in the presence of topology changes.

2. In nearly all cases, packets traveling through the Internet follow the geographically shortest route from their source to their destination.

   False. Packets typically travel through multiple networks using exchange points (or peering links) to switch networks, with little regard to geographic proximity. For instance, a packet from a Blacksburg residence might travel to Washington, D.C., switch networks, and come back to the Virginia Tech campus.

3. The public Internet consists not only of one network, but of multiple interconnected networks.

   True. These networks are called AS, or autonomous systems.

4. When designing modern web applications, the impact of propagation delay is negligible.

   False. The propagation delay provides a physical lower bound for round-trip time, and if applications require multiple round-trips these delays can quickly add up even if a single round-trip might incur imperceptible propagation delay. See 7 principles of rich web applications for more discussion.

5. If a packet-switched network becomes congested, then edge nodes will start rejecting new connections until the congestion subsides.

   False. Typical packet-switched networks do not maintain state about connections inside the network and do not perform admission control.

6. The term TCP/IP refers to the fact that applications using the IP network must also use TCP.

False. It's the other way around: TCP is designed to be used with IP, but other transport layer protocols (e.g., UDP) can be used with IP.

7. TCP can send data faster or slower depending on the bandwidth of the underlying network.

   True. That's the function of congestion control which will throttle a connection's speed as the available bandwidth in the network decreases and, conversely, send as fast as the network's bandwidth allows.

8. TCP authenticates users either using password-based or public-key based authentication before accepting a connection.

   False. TCP does not perform any kind of authentication.

9. An advantage of TCP is that it preserves message boundaries, which greatly simplifies the design of application-level protocols.

   False. TCP does not preserve message boundaries, placing the burden to reestablish them on application-level protocols.

10. In HTTP/1.1, clients must send a `Connection: close` header before they can close the connection.

    False. No, the connection close header is entirely optional/advisory. Both parties can close the connection at any point in time.

11. Web browsers such as Firefox or Chrome typically maintain multiple connections to each domain from which a website fetches objects.

    True. In this way, it's easier for servers to handle multiple requests in parallel, and typically more bandwidth can be used. However, clients must not create an excessive number of connections to obtain an unfair share of server resources. (For HTTP/2, the answer would be false as using a single connection is an explicit goal of HTTP/2.)

12. For the majority of HTTP transactions, the size of the response is greater than the size of the request (both including headers and bodies).

True. Typical requests do not have a body, and header sizes are small (and don't differ much between request/response). Overall, there is more traffic where HTTP servers provide objects than receive objects (as in the case of a file upload).

13. In a typical HTTP-based application, clients craft cookies using a cryptographic secret that is securely stored by a user agent such as a browser.

False. The cryptographic secret used to craft cookies must be stored on the server, not on the client. If it were accessible to the client, the client could craft cookies and impersonate other users.

14. Servers can detect if a client presents a JWT token that the client (or an intermediate) tampered with.

True. That's the purpose of the cryptographic signature that is used in these tokens.

1.2 Mixing Processes and Sockets (14 pts)

In a previous CS 3214 Final Exam, students were asked to find out what the following program does:

```c
#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);
```

```
20      if (rc != 0)
21          gai_strerror(rc), exit(EXIT_FAILURE);
22
23      while (info) {
24          int s = socket(info->ai_family,
25                         info->ai_socktype,
26                         info->ai_protocol);
27          if (s < 0)
28              perror("socket"), exit(EXIT_FAILURE);
29
30          if (connect(s, info->ai_addr, info->ai_addrlen) == 0)
31              return s;
32          close(s);
33      }
34      exit(EXIT_FAILURE);
35  }
36
37  struct fdpair {
38      int from, to;
39  };
40
41  static void *
42  shovel(void *_data)
43  {
44      struct fdpair * c = _data;
45      char buf[2048];
46      int bread;
47      while ((bread = read(c->from, buf, sizeof buf)) > 0)
48          write(c->to, buf, bread);
49      return NULL;
50  }
51
52  int
53  main(int ac, char *av[])
54  {
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 answer [URL] turned out to be that it worked as a slightly reduced variant of the netcat (nc) utility [URL].

In this question, you're given a companion program which unfortunately also lacks documentation

```
1   #include <stdlib.h>
2   #include <unistd.h>
3   #include <signal.h>
4   #include "socket.h"
5
6   int silent_mode;    // needed by socket.c
7
8   int
9   main(int ac, char *av[])
10  {
11      signal(SIGCHLD, SIG_IGN);
12      int socket = socket_open_bind_listen(av[1], 1024);
13      int client;
14
15      while ((client = socket_accept_client(socket)) != -1) {
16          if (fork() == 0) {
17              for (int fd = 0; fd < 3; fd++)
18                  dup2(client, fd);
19
20              execvp(av[2], av+2);
21              exit(0);
22          }
23          close(client);
24      }
25  }
```

This program uses the same `socket.c` and `socket.h` as in project 4.

1. (5 pts) 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 functional limitations you may spot as well (ignore the blatant lack of error checking which only for this example was elided intentionally for readability).

   [Solution]

```
USAGE:

./inetd port arg0 [arg1...argn] - start a remote network server

DESCRIPTION:
```

This program accepts two or more arguments, the first one being a port number of a TCP port to which the server will bind. The second argument is a Unix command. When a client connects to the server, a child process will be spawned that executes the command provided as argument, passing on any additional arguments to the command. All standard streams (standard input, standard output, and standard error) will be redirected to the accepted TCP connection.

2. (6 pts) How could you use it in conjunction with nc? Provide 2 useful examples.

   [Solution]

   Any program that communicates via its standard streams can work as an example, for instance,

   - `./inetd 2700 date` Provides a simple 'date' server - clients will receive the server's date.
   - `./inetd 2700 /bin/bash -i` Provides an interactive shell

3. (3 pts) Discuss the security implications of this tool.

   [Solution]

   As the second example above shows, this command must be used with care since it can give anyone connecting over the network direct access to the environment and privileges of the user executing the command. For most practical applications, an authentication mechanism would need to be added (such as spawning a login program instead of a shell). If a program is vulnerable and this vulnerability can be triggered via a specially crafted standard input, an attacker would have the ability to trigger this vulnerability remotely.

   In the early days of the Internet, so-called super servers (like 'inetd') were widely used to have a simple way to configure which network services a host provides in one point. These included services such as 'echo' which were widely enabled by default.

   However, because of rampant misconfiguration and lack of suitable access controls, such super servers are no longer in wide use today. Instead, by default a minimum amount of services is enabled, and administrators have to enable additional, separate services if needed.

2 Automatic Memory Management (18 pts)

2.1 Object Reachability Graphs (8 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. 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.

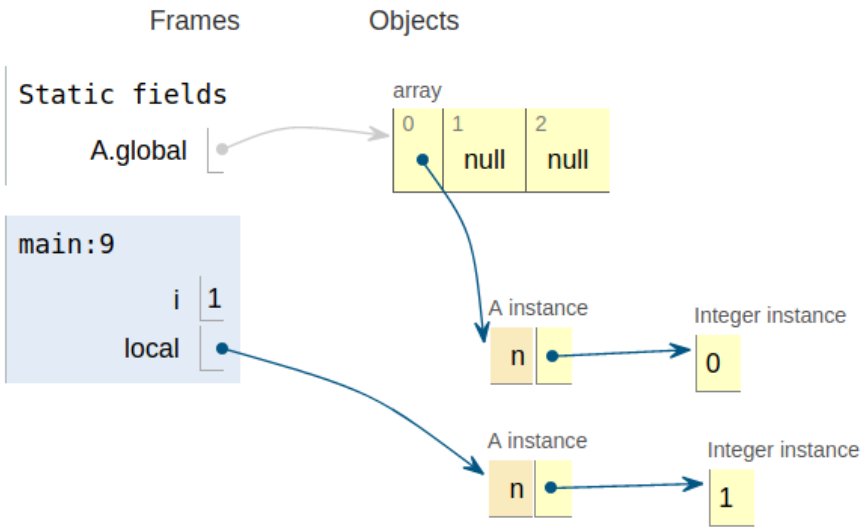

Figure 1: A snapshot of an object reachability graph produced by a Java program. On the left, roots are shown, with global static variables in white and stack frames in blue.

[Solution]

```
1  public class A {
2    static A [] global = new A[3];
3    Integer n;
4    A(int n) { this.n = n; }
5    public static void main(String[] args) {
6      A local;
7      for (int i = 0; i < global.length; i++) {
8        local = new A(i);  // here in 2nd iteration where i=1
9        global[i] = local;
10     }
```

```
11    }
12  }
```

The snapshot was taken during the 2nd iteration of the loop when the new object was assigned to `local`, but before it was assigned to `global[i]`. But any code that

- Allocates a 3-element global array
- Assigns global[0] to an A instance where n points to a java.lang.Integer of value 0
- Assigns a local variable to an A instance when n points to a java.lang.Integer of value 1
- Has a local variable i set to 1

would work. Note we did require the use of the boxed type as the difference between `int` and `Integer` is important to understand.

2.2 Cycles (5 pts)

According to Wikipedia's page on Garbage collection, Section Reference Counting, "if two or more objects refer to each other, they can create a cycle whereby neither will be collected as their mutual references never let their reference counts become zero."

Write a sample program in a garbage collected language of your choice that creates a cycle involving 3 garbage collectable objects.

[Solution]

Many programs are possible, including this example from a prior exam (if you've noticed)

```
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     public static void main(String[] args) {
12       Cycle C = new Cycle();
13       C = new Cycle(C);
14       C = new Cycle(C);
15       C.next.next.next = C;
```

```
16          C = null;  // make objects in cycle collectable
17      }
18  }
```

2.3 Leak Or Not (5 pts)

Consider the following Java program:

```
1   import java.util.*;
2   public class LeakOrNot
3   {
4       private int n;
5       LeakOrNot(int n) {
6           this.n = n;
7       }
8
9       static void work() {
10          // allocate
11          for (int i = 0; i < 1000; i++) {
12              set.add(new LeakOrNot(i));
13          }
14
15          // actual work elided
16
17          // cleanup
18          for (int i = 0; i < 1000; i++)
19              set.remove(new LeakOrNot(i));
20      }
21
22      static HashSet<LeakOrNot> set = new HashSet<>();
23  }
```

If the function `work()` were to be invoked repeatedly, would this program constitute a memory leak or not? Justify your answer. (You may use tools such as Eclipse Memory Analyzer or VisualVM if necessary.)

[Solution]

This program constitutes a leak because the `LeakOrNot` objects that are added to `set` in each call to `work)` are not, in fact, removed. The `set.remove` call is passed a new object that is not equal to any existing object in the collection, as per definition of `Object.equals()` and `Object.hashCode()`.

You can verify this by either checking the return value of `remove` and/or looking at the size of `set` after each call, or with a tool.

To make the code work in the intended way, the programmer would need to override and implement `Object.equals` and `Object.hashCode`. Failing to do is a common reason for memory leaks related to Java containers.

3 Virtual Memory (16 pts)

3.1 Files and Memory (8 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>
#include <sys/stat.h>

static char *
map_file(char *fname, off_t *size)
{
    int fd = open(fname, O_RDONLY);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    struct stat info;
    if (stat(fname, &info) == -1) {
        perror("stat");
        exit(EXIT_FAILURE);
    }

    char *p = mmap(NULL, *size = info.st_size,
                   PROT_READ, MAP_PRIVATE, fd, 0);
    if (p == NULL) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }
    return p;
}
```

```
32
33  int
34  main(int ac, char *av[])
35  {
36      if (ac != 3) {
37          fprintf(stderr, "Usage: %s file1 file2\n", av[0]);
38          return EXIT_FAILURE;
39      }
40
41      off_t asize, bsize;
42      char *a = map_file(av[1], &asize);
43      char *b = map_file(av[2], &bsize);
44      return (a && b && asize == bsize && memcmp(a, b, asize) == 0)
45              ? EXIT_SUCCESS : EXIT_FAILURE;
46  }
```

1. (3 pts) Come up with a suitable name for the program and 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 the exit status conventions of this utility.

   [Solution]

   ```
   USAGE:

   cmp file1 file2 - compare 2 files

   DESCRIPTION:
   ```

   This program accepts two command line arguments, file1 and file2 referring to 2 ordinary files. It will exit with 0 and indicate success if the two files are identical, it will exit with a non-zero exit code otherwise.

2. (3 pts) Provide an example of how to use this utility on the command line.

   [Solution]

   ```
   $ cp cmp.c copy.c
   $ if ./cmp copy.c cmp.c; then echo "identical"; else echo "different"; fi
   identical
   $ if ./cmp cmp.c cmp; then echo "identical"; else echo "different"; fi
   different
   ```

   or

```
$ ./cmp cmp.c cmp.c ; echo $?
0
$ ./cmp cmp.c cmp ; echo $?
1
```

Note that your example needed to include a way to examine the exit value (like if, or $?, or a boolean operator such as &&) - otherwise, since the program doesn't output anything, the user wouldn't be able to tell the result of the file comparison.

3. (2 pts) This program uses the `open(2)` system call, but not the `close(2)` system call, and it uses `mmap(2)`, but not `munmap(2)`. Does the omission constitute a defect in this program? Say why or why not.

   [Solution] No, it does not constitute a problem. The program exits, and upon exit the OS will close all file descriptors opened, and it will also destroy the process's virtual address space which includes unmapping any and all objects from it. There may be disagreement on whether it's good practice to close() and unmap() in programs designed for single runs, but it's clearly not a defect.

3.2 To Free or Not to Free (8 pts)

Dr. Back has been wondering if machines with relatively large memories, such as the node of our rlogin cluster, still require the use of `free()`. After all they come with 384 GB of memory so they should be able to accommodate a fair amount of waste. So he wrote the following optimized implementation of `free()`:

```
void free(void *ptr) { /* optimized out */ }
```

To test the implementation, he placed it into a shared library:

```
gcc -fPIC -Wall -c fastfree.c
gcc -shared -o libfastfree.so.1.0.1 fastfree.o
```

Then, to use the newly optimized `free()` with a program, this shared library will be preloaded so that its implementation is used instead of the standard `free()` implementation of the C library.

```
$ env LD_PRELOAD=./libfastfree.so.1.0.1 hostname
elm.rlogin
```

To benchmark his invention, he cloned the 'wrk' benchmarking tool from github and measured how long it would take to build it with the new implementation of `free()` and how long without it. Building this tool involves repeated invocation of the gcc compiler, the ld linker, and a number of smaller Unix utilities such as `install(1)`.

First, he tested without it:

```
$ cd /tmp
$ git clone https://github.com/wg/wrk.git wrk-gback
$ cd wrk-gback/
$ /usr/bin/time make -j 40
```

which concluded with the following output:

```
168.56user 27.63system 0:07.89elapsed 2484%CPU (0avgtext+0avgdata 106056maxresident)k
0inputs+0outputs (0major+7984270minor)pagefaults 0swaps
```

Then, he tested with the shared library that "optimizes" `free()`, like so:

```
$ make clean
$ env LD_PRELOAD=/home/staff/gback/tmp/fastmalloc/libfastfree.so.1.0.1 \
      /usr/bin/time make -j 40
```

which showed this result:

```
176.21user 69.88system 0:09.68elapsed 2540%CPU (0avgtext+0avgdata 609000maxresident)k
0inputs+0outputs (0major+24109164minor)pagefaults 0swaps
```

When he repeated the experiment a few times (running `make clean` in between runs), it showed similar results.

Your Task: analyze these results and provide a hypothesis that could explain them. Include in your analysis user time, system time, overall elapsed time, maxresident size and the number of minor pagefaults.[1]

[Solution] A reasonable hypothesis is the following.

First, the interpretation of the values: user time is the time spent in user mode, executing application code (such as the compiler/linker, etc.), including time spent in the user-level memory allocator (malloc/free). System time is time spent in kernel mode, including the time spent in memory-related system calls such as sbrk()/mmap(), and including the time spent servicing minor pagefaults. A minor pagefault occurs, among others, every time a virtual address inside a page allocated via sbrk()/mmap() is first accessed. The "max rss" size is the amount of physical memory that was allocated (because pages were touched) for the child or descendant for whom this number was highest. Here, make ran with 40 concurrent processes (`-j 40`) so this number, multiplied by 40, provides a good estimated upper bound on the maximum physical memory consumption of this workload.

[1] Note that the number of major pagefaults was 0 since Dr. Back worked inside the `/tmp` directory whose content is backed by RAM. If you experiment, please don't forget that /tmp is a shared resource. Delete any directories your create.

By omitting free(), allocated objects will never be returned to the heap. As a result, the user-level memory allocator will need to grow the heap by requesting more virtual addresses from the underlying OS (via sbrk() or mmap()) - just like in your p3 implementation you needed to call `mem_sbrk()` to expand the heap when there was nothing on the free lists. In a fully demand paged system like Linux, the OS does not allocate physical memory at this point. When the pages containing those additional virtual addresses are eventually accessed, minor pagefaults result, which explains the increase in the number of minor pagefaults and it explains the increase in the amount of resident (physical) memory used for the programs were free() was skipped. Note that because these machines have plenty of physical memory, the system never entered a state where it was unable to satisfy a request for physical memory when handling a page fault - the workload completed as normal. The minor page fault handler would simply request a free physical page from the kernel's memory allocator (a fast buddy allocator) and create a mapping to it.

The larger amount of time spent inside the kernel (system time) could be explained by at least 2 factors: (a) the time spent to handle these additional page faults, although this is unlikely to account for most of the increase, and (b) the additional time spent in the additional calls to sbrk() and mmap(), and (c) the increased amount of time spent when exiting each process (when the now larger virtual address space needs to be torn down and any physical memory that was mapped to it needs to be deallocated and returned to the physical memory allocator.)

Explaining the increase in user time is not as obvious: after all, the user programs executed by this workload do less than before, at least they save the time they would have otherwise spent in `free()`. A possible explanation may be an increase in cache and/or TLB misses due to the larger virtual memory footprint (along with an increase in time spent in the user portions of the code growing the heap, excluding the sbrk() and mmap() system calls).

While these are reasonable hypotheses that could explain the values observed, as in all of science, and in systems research in particular, we need to stress that those are hypotheses, not facts. A careful analysis and more extensive experimentation would be required to validate them, and it is not uncommon to discover other secondary or hidden factors that sometimes influence performance in unforeseen ways.

Incorrect Observations. There are also a number of incorrect observations some students made. First of, there is no "thrashing" with this workload. The total amount of physical memory used by the entire workload was no more than about 40×600MB, which is about 6.2% of the available memory in our rlogin nodes. There was never any risk of running out of physical memory (which would have necessitated swapping). This was also evident by the lack of major pagefaults which otherwise would be reported (in fact, there weren't major pagefaults due to any process-related activity, including the paging in of the executables).

Second, this workload did not "leak" any physical memory. There was a leaking of

allocated objects inside the virtual address space while each of the processes were running, requiring the OS to keep around virtual address allocations for the allocated but not freed objects, but at the end of the workload all processes had terminated, the OS had cleaned up after them, and the system had the same amount of usable physical memory as before. There were absolutely no persistent effects.

Third, be careful with the argument that `malloc()` will take longer if there are no freed objects in the heap because it spends "more time searching for objects." That's unlikely to be true: `malloc` will likely immediately tell that its free lists are empty and move to expand the heap. So to the extent that spending more time in malloc() contributes to the increase in user time, it's unlikely due to time being spent in the free list data structure lookup/find routines.

Fourth, I looked in your explanation for a clear distinction between the effects of managing memory blocks referring to virtual address ranges (as is done by the user-level malloc/free allocator) and the effects on virtual memory (where the OS allocates physical memory to service page faults). Notably, inside the OS, physical memory pages are also managed by an allocator, and this allocator needs to allocate memory on a page fault and return it to the free memory pool when a process exits, as discussed above.

Fifth, I read in a number of answers the formulation that a "minor pagefault" occurs when a "page is in memory" but not "allocated to the process." This definition appears to come from some online source, but it does not apply here. All additional minor page faults in this scenario are for anonymous memory (those are pages that the OS promised and will provide zero-filled pages on access which it must first allocate). In other words, there is no page that is already in memory. (This is unlike minor page faults that occur when accessing say the text segment of a shared library that is shared between different processes, or a process's text segment itself.)

4 Protection/Security (18 pts)

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <string.h>

#define MAX_SIZE sysconf(_SC_PAGE_SIZE)

char *processSpecificMessage = "Your message here";

int main(int argc, char **argv) {
```

```
13    // Get a page-aligned chunk of memory to store secrets
14    char * secretStash = (char *)mmap(NULL, MAX_SIZE, PROT_READ | PROT_WRITE,
15                                      MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
16    if(secretStash == MAP_FAILED) {
17      perror("ERROR: mmap failed: ");
18      return 1;
19    }
20
21    // Write my secrets
22    strncpy(secretStash, "SECRET MESSAGE", 15);
23    printf("PARENT: %s\n", secretStash);
24
25    // Hide the secret from my children and from core dump
26
27    // Prevent secret from going to swap
28
29    // Create children
30    for(int i = 1; i < argc; ++i) {
31      processSpecificMessage = argv[i];
32      pid_t cpid = fork();
33      if(!cpid) {
34        printf("CHILD %d: %s\n", i, secretStash);
35        printf("CHILD %d: %s\n", i, processSpecificMessage);
36        return 0;
37      }
38      wait(NULL);
39    }
40
41    // Prove parent still knows secret
42    printf("PARENT: %s\n", secretStash);
43  }
```

Analyze the program above, refer to the man page for `madvise`, and answer the following questions:

1. (1 pts) Without modification, do children know what is in their parent's `secretStash` at the time of their creation and why?

   [Solution] Yes, because children get a copy of their parent's memory at the time of their creation.

2. (1 pts) When, if ever, do children become unaware of what is in their parent's `secretStash`?

[Solution] Children only get a copy (see process isolation) at the time of creation, thus they cannot see their parent's updates to memory (see copy-on-write).

3. (2 pts) What statement would you add to line 26 to prevent children from knowing what is in their parent's `secretStash`?

   [Solution] See the solution code below (`MADV_WIPEONFORK`).

4. (2 pts) How can the child cause the parent's `secretStash` to be divulged through a core dump?

   [Solution] The child could send `SIGSEGV` signal to the parent, causing it to create a core dump.

5. (2 pts) What statement would you add on line 26 to prevent this (ignore error checking)?

   [Solution] See the solution code below (`MADV_DONTDUMP`).

6. (2 pts) How can the child cause the parent's `secretStash` to be divulged through a swap to disk?

   [Solution] Abstractly, swap occurs when there is insufficient physical memory to meet the demands of all processes on a system (i.e., the sum of their occupied virtual memory is greater than the available physical memory). For example, the child could `malloc()` and touch a large amount of memory, causing the parent's pages to be swapped-out to disk. See how this was used as part of a rowhammer attack in "Flip Feng Shui: Hammering a Needle in the Software Stack".

7. (2 pts) What statement would you add on line 28 to prevent this (ignore error checking)?

   [Solution] See the solution code below (`mlock(...)`).

8. (2 pts) Assuming all of the above modifications are in place, if the parent puts a second string midway in `secretStash`, i.e., `strcpy(secretStash + (MAX_SIZE >> 1), "Hello")`, what are the contents of that portion of `secretStash` from the child process's perspective and why?

   [Solution] Since `secretStash` is exactly one page in size and `madvise` works at page granularity, the entire page—as seen by the child—is cleared, i.e., set to 0, during creation.

9. (2 pts) Describe how you think the operating system implements the functionality used in Question 3 to ensure that children cannot see the secret message in their address space.

[Solution] Instead of copy on write, during child creation, the OS gives child processes fresh pages in place of the protected pages. Given that the MMU does not have specific flags for this, the OS must include flags to track these madvise features in each process's data structure.

10. (2 pts) Given that parents and children can run in any order, is there a risk that some children will have the same ID or message and why/why not?

    [Solution] No, because the data is not shared, only "copied" during child creation.

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <string.h>

#define MAX_SIZE sysconf(_SC_PAGE_SIZE)

char *processSpecificMessage = "Your message here";

int main(int argc, char **argv) {
  // Get a page-aligned chunk of memory to store secrets
  char * secretStash = (char *)mmap(NULL, MAX_SIZE, PROT_READ | PROT_WRITE, MAP_PRIVATE
                                    | MAP_ANONYMOUS, -1, 0);
  if(secretStash == MAP_FAILED) {
    perror("ERROR: mmap failed: ");
    return 1;
  }

  // Write my secrets
  strncpy(secretStash, "SECRET MESSAGE", 15);
  printf("PARENT: %s\n", secretStash);

  // Hide the secret from my children and from core dump
  int ret = madvise(secretStash, MAX_SIZE, MADV_WIPEONFORK | MADV_DONTDUMP);
  if(ret) {
    perror("ERROR: madvise: ");
    return 1;
  }

  // Prevent secret from going to swap
```

```
33    if(mlock(secretStash, MAX_SIZE)) {
34      perror("mlock: ");
35      return 2;
36    }
37
38    // Create children
39    for(int i = 1; i < argc; ++i) {
40      processSpecificMessage = argv[i];
41      pid_t cpid = fork();
42      if(!cpid) {
43        printf("CHILD %d: %s\n", i, secretStash);
44        printf("CHILD %d: %s\n", i, processSpecificMessage);
45        return 0;
46      }
47      wait(NULL);
48    }
49
50    // Prove parent still knows secret
51    printf("PARENT: %s\n", secretStash);
52  }
```

5 Dynamic Memory Management (20 pts)

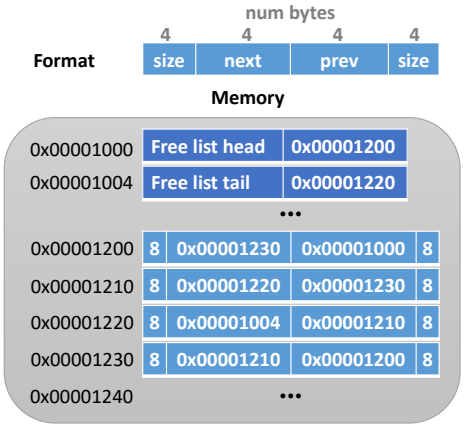

Figure 2: Free list in memory. Assume the free list management policies are Best-fit for allocation, freed blocks are added to the front of the list, and coalesced when needed to service a `malloc()`. You may ignore splitting.

Consider the following code snippet:

```
1  // Question 1
2  void * a = malloc(8);
3  void * b = malloc(8);
4  void * c = malloc(8);
5
6  // Question 3
7  free(c);
8  free(a);
9  free(b);
10
11 // Question 5
12 a = malloc(15);
13
14 // Question 6
15 free(a);
16
17 // Question 7
18 *((char *)b) = "DEADBEEF";
19
20 // Question 8
21 a = malloc(8);
22
23 // Question 9
24 uint32_t x = *((uint32_t *)a);
```

1. (2 pts) Given the free list pictured in Figure 2, list the values of variables `a`, `b`, and `c` after executing up to line 5?

   [Solution] a = 0x00001204, b = 0x00001234, and c = 0x00001214. Keep in mind that the start of a free list entry is size metadata, which is 4 bytes.

2. (1 pts) What does the free list look like at line 5? Report your answer in the format: head address, tail address(, size, next, prev, size)*.[2] Please use hexadecimal notation for each value and you may remove leading zeros.

   [Solution] 0x1220, 0x1220, 0x8, 0x1004, 0x1000, 0x8. Note that it is also okay to surround the entry list with parenthesis.

3. (1 pts) What does the free list look like after executing up to line 10 (use the same format as the previous question)?

[2] (...)* means that the sequence in the parenthesis can appear zero or more times

[Solution] 0x1230, 0x1220, 0x8, 0x1200, 0x1000, 0x8, 0x8, 0x1210, 0x1230, 0x8, 0x8, 0x1220, 0x1200, 0x8, 0x8, 0x1004, 0x1210, 0x8. Note that the entries may appear in memory order or in list order, and may or may not have parenthesis.

4. (2 pts) In general, does the order that `free()`'s occur in matter? Why?

   [Solution] Yes, because, depending on the allocator policies implemented, it influences the run-time performance of the allocator.

5. (2 pts) What address is returned by the call to `malloc()` on line 12 and what is the state of the free list after this call?

   [Solution] Servicing this request requires coalescing. If you assume forward-only coalescing: a = 0x1204 and free list = 0x1230, 0x1220, 0x8, 0x1004, 0x1230, 0x8, 0x8, 0x1220, 0x1000, 0x8. If you assume backward coalescing: a= 0x1224 and free list = 0x1200, 0x1210, 0x8, 0x1210, 0x1000, 0x8, 0x8, 0x1004, 0x1200, 0x8.

6. (1 pts) What does the free list look like after the call to `free()` on line 15?

   [Solution] Assuming forward coalescing in previous response: 0x1200, 0x1220, 0x18, 0x1230, 0x1000, 0x18, 0x8, 0x1004, 0x1230, 0x8, 0x8, 0x1220, 0x1200, 0x8. Assuming backward coalescing in previous response: 0x1220, 0x1210, 0x8, 0x1210, 0x1220, 0x8, 0x8, 0x1004, 0x1200, 0x8, 0x18, 0x1200, 0x1000, 0x18.

7. (2 pts) What are the contents of the free list after executing the assignment statement on line 18?

   [Solution] Assuming forward coalescing in previous response: 0x1200, 0x1220, 0x18, 0x1230, 0x1000, 0x18, 0x8, 0x1004, 0x1230, 0x8, 0x8, 0x44454144, 0x42454546, 0x8. Assuming backward coalescing in previous response: 0x1220, 0x1210, 0x8, 0x1210, 0x1220, 0x8, 0x8, 0x1004, 0x1200, 0x8, 0x18, 0x1200, 0x42454546, 0x18.

8. (2 pts) What is the value of variable `a` after executing line 21? Why?

   [Solution] Assuming forward coalescing in previous response: 0x1234. Assuming backward coalescing in previous response: 0x1204.

9. (2 pts) If we now execute up through line 24, what is the value of `x` (unknown is a viable answer)?

   [Solution] Assuming forward coalescing in previous response: 0x44454144. Assuming backward coalescing in previous response: 0x00001210.

10. (3 pts) Put together everything you know to complete the attack below (the details do not need to be exact for full credit): given the password checking code below, how would you cause the program to execute `giveStudentA()`, even though the password does not match? You may add exactly one statement to line 11 to finish

the attack. This statement is limited to using `malloc()`, `free()`, variable `a` (as an assignment target, i.e., `*a = ...`), and you may use `passwordMatches` on the right-hand side of the statement. The target of the attack is the simple free list implementation developed in the earlier questions. As with the previous code snippet, memory addresses are 32-bits (hence using the `uint32_t` data type).

```
uint32_t passwordMatches = 0; // non-0 when password is correct

uint32_t * a;

// Start with empty free list

// Put something on the free list
a = (uint32_t *)malloc(8);
free(a);

// YOUR one statement of code here - copy to allocator.txt

a = (uint32_t *)malloc(8);

// Get a pointer to passwordMatches
a = (uint32_t *)malloc(8);

// Change value of passwordMatches
*a = 1;

if(passwordMatches) {
  giveStudentA();
}
```

[Solution] This is an example of a free list poisoning attack. The key idea is to trick the allocator into giving you access to memory that you wouldn't normally have access to; in this case, the result of the password check stored in `passwordMatches`. You do this by poisoning the list pointers in a previously-freed pointer with the address of the victim variable. While the exact line is allocator dependent, it follows the form:

`*a = (uint32_t)(&passwordMatches - SIZE_BYTES);`.

11. (2 pts) How would you, as a dynamic memory allocator designer, protect against this attack?

    [Solution] There are a variety of acceptable (as answers) ways to do this that vary in complexity and run-time overhead. One approach is heap hardening as used in

Linux. In heap hardening, the pointers used by free list entries are masked with unpredictable values at run time.