CS 3214 Spring 2021 Final Exam Solution

September 13, 2021

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.

1 Virtual Memory (26 pts)

1.1 Understanding Page Faults (6 pts)

Like most modern OSes, Linux uses fully on-demand paged virtual memory. The following shell script uses the `time(1)` command to display the number of minor 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 this script on an empty C program (which includes only the starter code that calls main()), i.e.:

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

yields an output of about 50–60 pagefaults on a contemporary Linux x86_64 machine with a 4 KB page size.

Change the following program so that it produces apprx. 562–572 minor pagefaults:

```
#include <stdlib.h>
#define M 256
#define S 4096

char global[M*S];

void *f()
{
    int i;
```

```
    char * m = malloc(M * S);
    return m;
}

int
main()
{
    int i;

    f();
}
```

Rules:

- You **may not** introduce any new variable definitions of any kind, local or global variables.
- You **may not** introduce any more calls to `malloc()` or other memory allocation functions.
- You **may not** call `fork()`.

Solution: Any code that touches the 256 pages returned by the malloc call, as well as the 256 pages that constitute the variable "global" will do, e.g.

```
#include <stdlib.h>
#define M 256
#define S 4096

char global[M*S];

void *f()
{
    int i;
    char * m = malloc(M * S);
#ifdef SOLUTION
    for (i = 0; i < M; i++)
        m[i*S] = 1;
#endif
    return m;
}

int
```

```
main()
{
    int i;

#ifdef SOLUTION
    for (i = 0; i < M; i++)
        global[i*S] = 1;
#endif

    f();
}
```

1.2 Files and Memory (12 pts)

Consider the following Unix program that unfortunately lacks documentation.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <fcntl.h>
6  #include <sys/mman.h>
7  #include <sys/stat.h>
8
9  static char *
10 map_file(char *fname, off_t *size)
11 {
12     int fd = STDIN_FILENO;
13     if (strcmp(fname, "-") != 0)
14         fd = open(fname, O_RDONLY);
15
16     if (fd == -1) {
17         perror("open");
18         exit(EXIT_FAILURE);
19     }
20
21     struct stat info;
22     if (fstat(fd, &info) == 1) {
23         perror("stat");
24         exit(EXIT_FAILURE);
25     }
```

```
26
27      char *p = mmap(NULL, *size = info.st_size,
28                     PROT_READ, MAP_PRIVATE, fd, 0);
29      if (p == MAP_FAILED) {
30          perror("mmap");
31          exit(EXIT_FAILURE);
32      }
33      return p;
34  }
35
36  int
37  main(int ac, char *av[])
38  {
39      off_t asize;
40      char *a = map_file(ac > 1 ? av[1] : "-", &asize);
41      char *l = a + asize - 1;
42      for (char *p = l; a <= p; p--) {
43          if (*p == '\n' && l != p) {
44              write (STDOUT_FILENO, p + 1, l - p);
45              l = p;
46          }
47      }
48      write (STDOUT_FILENO, a, l - a + 1);
49  }
```

1. (2 pts) Write a brief, man-page like description of what this utility program does. Be sure to include usage instructions and a synopsis of the intended function.

   Solution:

   USAGE:

   ./tac [filename|-] - print file in reverse

   DESCRIPTION:

   If an argument is given, this program opens the file "filename" and outputs the lines contained in it in reverse. If - is given as the filename, or if no filename is given, the program reverses the lines in its standard input.

   NOTES:

The program will fail if its standard input stream does not refer to an mmappable file.

2. (2 pts) Provide an example of how to use this utility on the command line without using I/O redirection.

   Solution:

   ```
   ./tac textfile.txt
   ```

3. (2 pts) Does this utility work on the command line when its standard input is redirected from a file? Briefly justify why or why not.

   Solution: Yes. The shell opens the file and arranges for file descriptor 0 to refer to it. This is indistinguishable from the case where the `tac` program opens the file, hence mmapping its content will work in both cases.

4. (2 pts) Can the program be used at the receiving end of a pipeline? Show an example or justify why not.

   Solution: No, it cannot. The standard input stream of a program at the receiving end of a pipeline is a pipe object, which does not support mmap() since it is not backed by a file or other mmappable object.

5. (2 pts) Can the program be used at the sending end of a pipeline? Show an example or justify why not.

   Solution: Yes, for instance `./tac textfile.txt | wc`

6. (2 pts) Find the name of the existing Linux utility that this program resembles.

   Solution: The name is `tac` (`cat` reversed), written by Jay Lepreau and David McKenzie. Lepreau served on Dr. Back's doctoral advisory committee.

1.3 Linus Torvalds on Shared Libraries (8 pts)

In a recent post on the Linux kernel developer mailing list, Linus Torvalds opined that "Shared libraries are not a good thing in general."

The full quote ([URL]) is provided below. For context, the discussion ensued when Torvalds noticed that the clang compiler takes a long time to start up because of its need to dynamically load the llvm library, which is required to be linked dynamically by the Fedora Linux distribution's policies.

Torvalds wrote:

> Shared libraries are not a good thing in general. They add a lot of overhead in this case, but more importantly they also add lots of unnecessary dependencies

> and complexity, and almost no shared libraries are actually version-safe, so it adds absolutely zero upside.
>
> Yes, it can save on disk use, but unless it's some very core library used by a lot of things (ie particularly things like GUI libraries like gnome or Qt or similar), the disk savings are often not all that big - and disk is cheap. And the memory savings are often actually negative (again, unless it's some big library that is typically used by lots of different programs at the same time).
>
> In this case, for example, it's true that a parallel build will be running possibly hundreds of copies of clang at the same time - and they'll all share the shared llvm library. But they'd share those same pages even if it wasn't a shared library, because it's the same executable! And the dynamic linking will actually cause a lot _less_ sharing because of all the fixups.
>
> We hit this in the subsurface project too. We had a couple of libraries that *nobody* else used. Literally *nobody*. But the Fedora policy meant that a Fedora package had to go the extra mile to make those other libraries be shared libraries, for actual negative gain, and a much more fragile end result (since those libraries were in no way compatible across different versions - so it all had to be updated in lock-step).
>
> I think people have this incorrect picture that "shared libraries are inherently good". They really really aren't. They cause a lot of problems, and the advantage really should always be weighed against those (big) disadvantages.
>
> Pretty much the only case shared libraries really make sense is for truly standardized system libraries that are everywhere, and are part of the base distro.

This question focuses on two statements Torvalds makes, rephrased below as follows:

1. (4 pts) When there are many copies of a program that use the same library, then this library's code would be shared even if it were statically linked into the executable.

2. (4 pts) When there are "truly standardized system libraries" or "very core libraries" like gnome or Qt, then having shared libraries make sense because they save memory.

Draw **two** sketches that illustrate Torvalds' statements, one for each of these two cases. Your sketches should depict the processes' virtual address space(s) and physical memory in sufficient detail in order to clearly depict the described scenarios.

Solution: The possible sketches are shown in Figures 1 and 2, respectively.

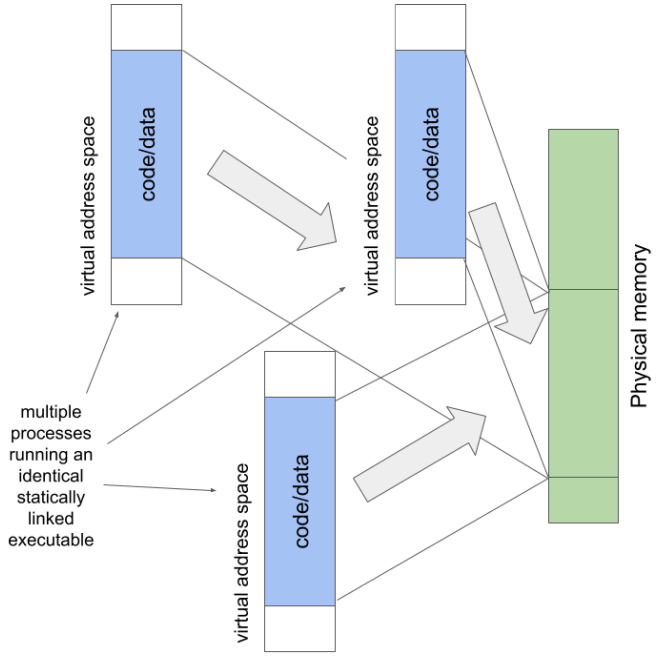


Figure 1: Multiple statically linked executables sharing physical memory.

2 Networking (38 pts)

2.1 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. Internet core routers can connect networks belonging to different organizations.

   True

2. When packets travel through the Internet, the source host computes the route they should take to their destination.

   False The Internet does not use source routing; instead, each router determines the next hop a packet takes based on its forwarding table.

3. The Internet's core routers provide a reliable packet delivery service for end hosts.

   False The IP protocol provides an unreliable, best-effort packet delivery to end hosts.

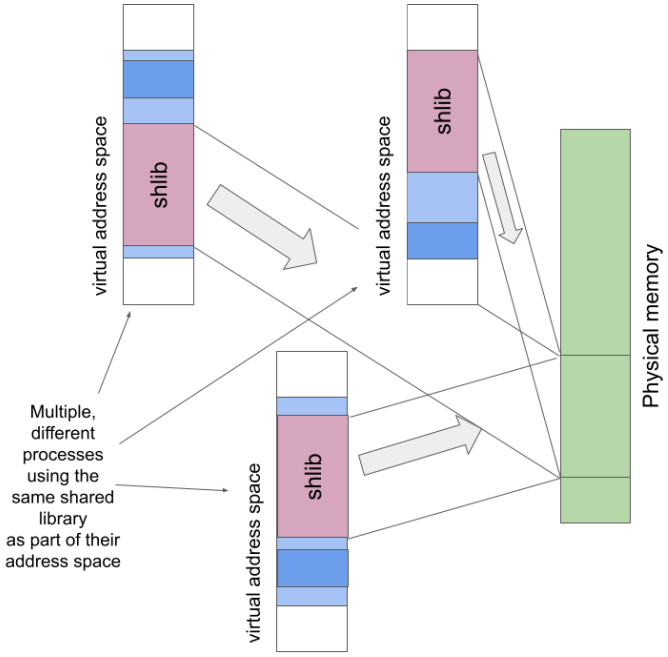


Figure 2: Multiple dynamically linked executables sharing physical memory when a common shared library is used.

4. To communicate on the Internet, a host must have exclusive or shared use of a public IP address assigned by a global entity.

   True

5. When using TCP, the communicating parties must exchange control messages before a connection can be established.

   True

6. The TCP protocol traditionally anticipates packet losses in unreliable networks by sending multiple, redundant copies of a packet to the destination.

   False. TCP traditionally relies on acknowledgements and retransmission to overcome packet losses in unreliable networks. (Forward error correction is a possible extension, but not part of traditional TCP.)

7. When a process crashes, the OS will close any TCP connections that were used exclusively by that process automatically.

True

8. A load balancer such as `http://video2loadbalancer-1343016661.us-east-1.elb.amazonaws.com/` must examine the client's `Host:` header to know where to route the request.

   True

9. The server running load balancers such as `http://video2loadbalancer-1343016661.us-east-1.elb.amazonaws.com/` is most likely written as an event-based server.

   True

10. Modern web applications often use a design that logically separates static HTML and JavaScript code from the user data in terms of representation and transfer.

    True

2.2 Mysterious Streaming Forces (6 pts)

In Project 4 you implemented a video streaming service. When the client sent a request with a `Range:` header such as `Range: bytes=0-`, your server responded with the appropriate headers and then the video content started streaming to the client. "Streaming" here means to send just enough data to the client to render the video for a human user. For instance, a typical 1080p video might be sent at a speed of 4-5 Mbps/s.

After determining the range the client requested, the server started streaming using this loop (which is part of the p4 basecode):

```
// assume from, to have been set to refer to begin/end offset of range
while (success && from <= to)
  success = bufio_sendfile(ta->client->bufio, filefd, &from, to+1-from) > 0;
```

Explain in 3-4 sentences why the server did not need to determine the video's bitrate in order to compute how much data to send per time unit? Refer to your knowledge of network protocols as needed.

Solution:

The server does not need to determine the bitrate because the client will receive only as many bytes per time unit as it needs to play the video for the user. TCP flow control ensures that a sender is slowed down and (if necessary) blocked if the receiver's speed is less than the sender's.

(Additional detail, not needed for a correct answer:) When the receive window is empty, the sender is in the BLOCKED state. Once the recipient picks up the data with a recv or read system call, the recipient's OS kernel will advertise more space in the receive window by sending a message to the sender's host. This will allow the sender to send more data.

2.3 Alternative Tokens (12 pts)

After implementing Project 4, your project partner reflected on the use of JWT tokens with HS256 signatures with the following statements. State whether they are **true** or **false**.

1. (True/False, 2 pts) Each JWT token contains a payload with claims such as the subject ("sub"), issue and expiration time, in plain text that is Base64-encoded.

   True

2. (True/False, 2 pts) The server presents such a JWT token to a client as proof of a prior successful authentication.

   True

3. (True/False, 2 pts) The client cannot create a JWT token that the server accepts by itself because the server includes a randomly chosen string value that functions as a kind of "signature" to the token.

   False The signature is not a randomly chosen string value. It is computed by applying an HMAC algorithm to the JWT token's payload combined with a secret key.

4. (True/False, 2 pts) Once a client obtains a token, the server cannot easily revoke the token until it expires.

   True

Your partner then proposes an alternative design that works as follows. The server uses a secret seed (a number) determined on startup, which is passed to the standard `srand()` C function, and assign subsequent random numbers returned by the `rand()` function to create simpler cookies that look like this:

```
auth_token=34ab43bcde82f234_user0
```

where 0x34ab43bcde82f234 is the unique random number assigned and where '_user0' substitutes for the account id of the user. In this scheme, the next user may get a cookie such as:

```
auth_token=7f34c83945ea40ff2_user1
```

Then, to implement expiration, the server stores the as-of-now unexpired tokens in a linked hash table, evicting those whose validity has expired as time progresses. The hash table design ensures that presented tokens can be quickly looked up. If a user's token needs to be revoked, it can be removed from the hash table immediately.

(4 pts) Briefly discuss the merits of this proposal, specifically

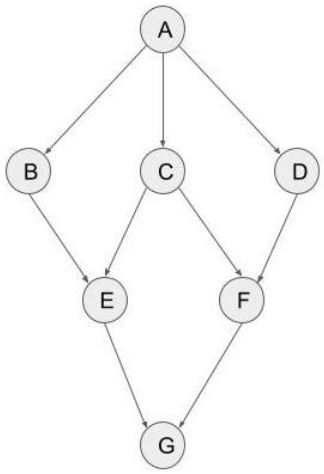
- Does it provide secure authentication?

- If so, briefly justify why. If not, state how it can be corrected.

Solution: This approach does not provide secure authentication because it does not use a cryptographically secure function to compute the authentication tokens. In other words, it is relatively easy for attackers to learn the values assigned to other users' token if a few elements of the random number sequence are known.

The scheme can, however, be fixed by using a cryptographically secure random number generator. In that case, the proposed scheme is essentially session-based authentication in which a client obtains an unforgeable session token which is presented to the server and which the server looks up in a table upon receipt to check its validity.

2.4 Map Reduce, Take 3 (10 pts)



CS3214 this semester would not be complete without another take on the MapReduce example that has accompanied us since Test 1.

In this round, we will create a version that uses TCP/IP sockets. We provide the scaffolding, and your task is to identify and add the functions necessary to create the TCP/IP connections.

You may assume that the program will be linked with an updated version of `socket.c` (available on the website) which includes a new function:

```
// create a TCP/IP connection to host 'hostname', port 'port_number_string'
// on success, return a new socket
// on failure, return -1 and set errno
int socket_connect(const char *hostname, const char *port_number_string);
```

This, the program will be built with:

`gcc mapreduce-with-sockets.c socket.c -o mr`

and it will be run with:

`env PORT=10203 ./mr`

where we will replace `10203` with a suitable number n. You may assume that no other server uses ports in the range $[n, n + 10]$.

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

#define PROCS 7
#define TOTAL 100
#define MSG_SIZE 20

static char *portbase;
/* add any functions you need here */

static int a_to_b, a_to_c, a_to_d, c_to_e, c_to_f, total;

/* Read `total` messages from `fromfd` and
 * round-robin distribute to sockets at `(tofd + i) % stride`
 *
 * For simplicity, we send always 20-byte messages padded with 00 bytes.
 * If `pad` is false, we do not pad.
 */
static void process(const char *name, int fromfd, int tofd[],
                    int total, int stride, bool pad)
{
  char msgi[MSG_SIZE], msgo[MSG_SIZE];
  memset(msgo, 0, sizeof msgo);     // avoid valgrind warning

  for (int count = 0; count < total; count++) {
    read(fromfd, msgi, MSG_SIZE);
    snprintf(msgo, sizeof msgo, "%s%s", name, msgi);
    write(tofd[count%stride], msgo, pad ? MSG_SIZE : strlen(msgo));
```

```
  }
}

static int divceil(int x, int y)
{
  return (x + y - 1) / y;
}

static void task_a()
{
  // IMPLEMENT THIS
  // process("A", src, dst, total, 3, true);
}

static void task_b()
{
  // IMPLEMENT THIS
  // process("B", src, dst, a_to_b, 1, true);
}

static void task_c()
{
  // IMPLEMENT THIS
  // process("C", src, dst, a_to_c, 2, true);
}

static void task_d()
{
  // IMPLEMENT THIS
  // process("D", src, dst, a_to_d, 1, true);
}

static void task_e()
{
  // IMPLEMENT THIS
  // process("E", src1, dst, a_to_b, 1, true);
  // process("E", src2, dst, c_to_e, 1, true);
}

static void task_f()
{
```

```
75    // IMPLEMENT THIS
76    // process("F", src1, dst, c_to_f, 1, true);
77    // process("F", src2, dst, a_to_d, 1, true);
78  }
79
80  static void task_g()
81  {
82    // IMPLEMENT THIS
83    // int stdio[1] = { 1 };
84    // process("G", src1, stdio, a_to_b + c_to_e, 1, false);
85    // process("G", src2, stdio, c_to_f + a_to_d, 1, false);
86  }
87
88  int
89  main(int ac, char *av[])
90  {
91    // compute how many messages will be sent
92    total = ac > 1 ? atoi(av[1]) : TOTAL;
93    a_to_b = divceil(total, 3);
94    a_to_c = divceil(total-1, 3);
95    a_to_d = divceil(total-2, 3);
96    c_to_e = divceil(a_to_c, 2);
97    c_to_f = divceil(a_to_c-1, 2);
98
99    portbase = getenv("PORT");
100
101   // let's create the processes next
102   for (int i = 0; i < PROCS; i++)
103     if ((fork()) == 0) {
104       switch ('A' + i) {
105         case 'A': task_a(); break;
106         case 'B': task_b(); break;
107         case 'C': task_c(); break;
108         case 'D': task_d(); break;
109         case 'E': task_e(); break;
110         case 'F': task_f(); break;
111         case 'G': task_g(); break;
112       }
113       exit(EXIT_SUCCESS);
114     }
115
```

```
116   // allow 1s for all mappers/reducers to start up
117   sleep(1);
118   int entry = -1;   // IMPLEMENT THIS
119
120   // send numbers to A process
121   for (int i = 0; i < total; i++) {
122     char msg[MSG_SIZE];
123     memset(msg, 0, sizeof msg);     // avoid valgrind warning
124     snprintf(msg, sizeof msg, "%d\n", i);
125     write(entry, msg, MSG_SIZE);
126   }
127
128   for (int i = 0; i < PROCS; i++)
129     wait(NULL);
130 }
```

Solution:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4  #include <string.h>
5  #include <unistd.h>
6  #include <sys/wait.h>
7  #include "socket.h"
8
9  #define PROCS 7
10 #define TOTAL 100
11 #define MSG_SIZE 20
12
13 static char *portbase;
14 static char * getport(int i)
15 {
16     char * b = malloc(10);
17     snprintf(b, 10, "%d", atoi(portbase) + i);
18     return b;
19 }
20
21 static int a_to_b, a_to_c, a_to_d, c_to_e, c_to_f, total;
22
23 /* Read `total` messages from `fromfd` and
```

```
 * round-robin distribute to sockets at `(tofd + i) % stride`
 *
 * For simplicity, we send always 20-byte messages padded with 00 bytes.
 * If `pad` is false, we do not pad.
 */
static void process(const char *name, int fromfd, int tofd[],
                    int total, int stride, bool pad)
{
  char msgi[MSG_SIZE], msgo[MSG_SIZE];
  memset(msgo, 0, sizeof msgo);     // avoid valgrind warning

  for (int count = 0; count < total; count++) {
    read(fromfd, msgi, MSG_SIZE);
    snprintf(msgo, sizeof msgo, "%s%s", name, msgi);
    write(tofd[count%stride], msgo, pad ? MSG_SIZE : strlen(msgo));
  }
}

static int divceil(int x, int y)
{
  return (x + y - 1) / y;
}

static void task_a()
{
  int asrc = socket_open_bind_listen(portbase, 1);
  int src = socket_accept_client(asrc);
  int dst[3] = {
      socket_connect("localhost", getport(1)),
      socket_connect("localhost", getport(2)),
      socket_connect("localhost", getport(3))
  };
  process("A", src, dst, total, 3, true);
}

static void task_b()
{
  int asrc = socket_open_bind_listen(getport(1), 1);
  int src = socket_accept_client(asrc);
  int dst[1] = { socket_connect("localhost", getport(4)) };
  process("B", src, dst, a_to_b, 1, true);
```

```
65  }
66
67  static void task_c()
68  {
69    int asrc = socket_open_bind_listen(getport(2), 1);
70    int src = socket_accept_client(asrc);
71    int dst[2] = {
72        socket_connect("localhost", getport(5)),
73        socket_connect("localhost", getport(6)),
74    };
75    process("C", src, dst, a_to_c, 2, true);
76  }
77
78  static void task_d()
79  {
80    int asrc = socket_open_bind_listen(getport(3), 1);
81    int src = socket_accept_client(asrc);
82    int dst[1] = { socket_connect("localhost", getport(7)) };
83    process("D", src, dst, a_to_d, 1, true);
84  }
85
86  static void task_e()
87  {
88    int asrc1 = socket_open_bind_listen(getport(4), 1);
89    int asrc2 = socket_open_bind_listen(getport(5), 1);
90    int src1 = socket_accept_client(asrc1);
91    int src2 = socket_accept_client(asrc2);
92    int dst[1] = { socket_connect("localhost", getport(8)) };
93    process("E", src1, dst, a_to_b, 1, true);
94    process("E", src2, dst, c_to_e, 1, true);
95  }
96
97  static void task_f()
98  {
99    int asrc1 = socket_open_bind_listen(getport(6), 1);
100   int asrc2 = socket_open_bind_listen(getport(7), 1);
101   int src1 = socket_accept_client(asrc1);
102   int src2 = socket_accept_client(asrc2);
103   int dst[1] = { socket_connect("localhost", getport(9)) };
104   process("F", src1, dst, c_to_f, 1, true);
105   process("F", src2, dst, a_to_d, 1, true);
```

```
106 }
107
108 static void task_g()
109 {
110   int asrc1 = socket_open_bind_listen(getport(8), 1);
111   int asrc2 = socket_open_bind_listen(getport(9), 1);
112   int src1 = socket_accept_client(asrc1);
113   int src2 = socket_accept_client(asrc2);
114   int stdio[1] = { 1 };
115   process("G", src1, stdio, a_to_b + c_to_e, 1, false);
116   process("G", src2, stdio, c_to_f + a_to_d, 1, false);
117 }
118
119 int
120 main(int ac, char *av[])
121 {
122   // compute how many messages will be sent
123   total = ac > 1 ? atoi(av[1]) : TOTAL;
124   a_to_b = divceil(total, 3);
125   a_to_c = divceil(total-1, 3);
126   a_to_d = divceil(total-2, 3);
127   c_to_e = divceil(a_to_c, 2);
128   c_to_f = divceil(a_to_c-1, 2);
129
130   portbase = getenv("PORT");
131
132   // let's create the processes next
133   for (int i = 0; i < PROCS; i++)
134     if ((fork()) == 0) {
135       switch ('A' + i) {
136         case 'A': task_a(); break;
137         case 'B': task_b(); break;
138         case 'C': task_c(); break;
139         case 'D': task_d(); break;
140         case 'E': task_e(); break;
141         case 'F': task_f(); break;
142         case 'G': task_g(); break;
143       }
144       exit(EXIT_SUCCESS);
145     }
146
```

```
147   sleep(1);
148   int entry = socket_connect("localhost", portbase);
149
150   // send numbers to A process
151   for (int i = 0; i < total; i++) {
152     char msg[MSG_SIZE];
153     memset(msg, 0, sizeof msg);     // avoid valgrind warning
154     snprintf(msg, sizeof msg, "%d\n", i);
155     write(entry, msg, MSG_SIZE);
156   }
157
158   for (int i = 0; i < PROCS; i++)
159     wait(NULL);
160 }
```

3 Automatic Memory Management (16 pts)

3.1 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 3 shows a snapshot of a heap produced by the execution of a small Java program.

Reconstruct this program, and denote with a comment the point in time at which the heap has the structure displayed in Figure 3. Assume that no garbage collection has taken place.[1]

[1]Note that it is not possible to create the figure as presented with the Java tutor at http://pythontutor.com/java.html

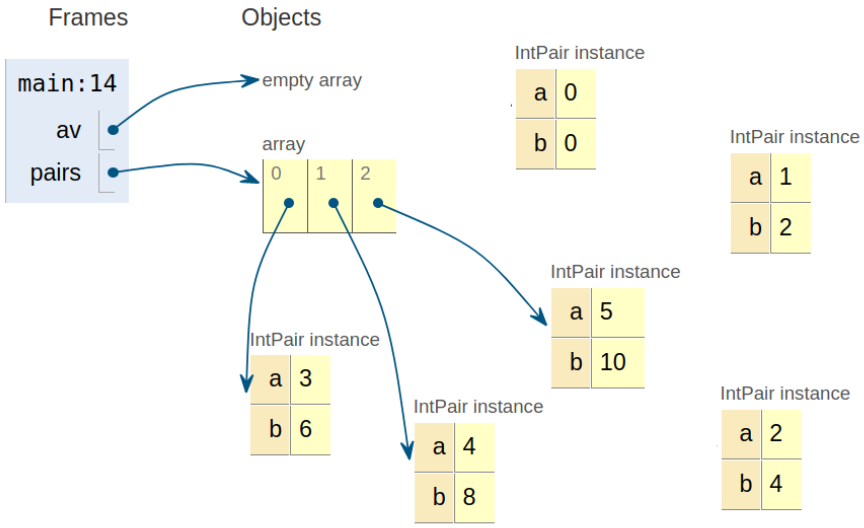


Figure 3: A snapshot of a heap produced by a Java program. On the left, roots are shown, with stack frames marked in blue. The heap includes unreachable objects as well.

Solution:

```
public class Object {
    static class IntPair {
        int a, b;
        IntPair(int a, int b) {
            this.a = a;
            this.b = b;
        }
    }
    public static void main(String []av) {
        IntPair [] pairs = new IntPair[3];
        for (int i = 0; i < 6; i++) {
            pairs[i%pairs.length] = new IntPair(i, 2*i);
        }
    }
}
```

3.2 Debugging Memory Profiles (8 pts)

In a 2019 Medium post titled "Hunting for Memory Leaks in Python applications" [URL] Wai Chee Yau, an engineer at Zendesk, discusses how to deal with memory-related perfor-

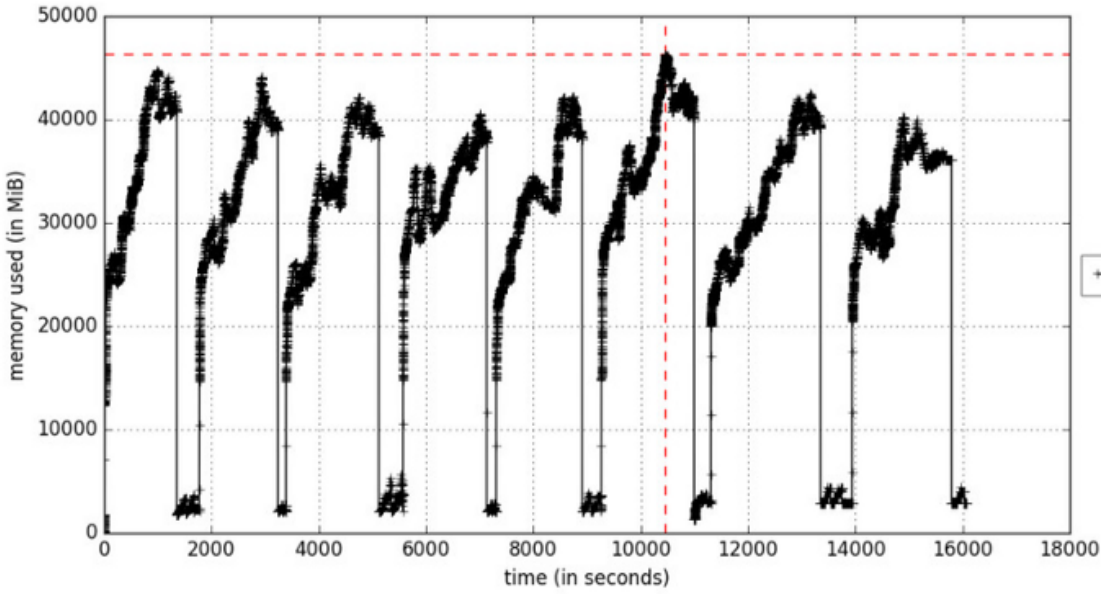


Figure 4: Memory profile as a function of time. Source: Wai Chee Yau, ZenDesk

mance issues encountered in some machine learning applications in use at Zendesk. This article contained the memory profile shown in Figure 4.

Answer the following questions:

1. (4 pts) The chart shows allocated memory, but does not show a profile of live memory. Use a paint program that allows you to make and save annotations to the chart. Annotate the chart to show a possible curve for live memory that is consistent with the information provided.

   Solution:

   We do not have any information about the live heap size, other than that it's strictly below the allocated memory at all points in its execution. It could suffer from any combination of churn, bloat, or simply tackling a large problem but do so in a manner that's efficient. We do not see any significant leaks since the memory utilization goes down to a low level after each phase of computation. We show two possibilities here that would lend themselves to being classified as churn and (potentially) bloat.

   Figure 5 shows a (theoretical) profile where the live heap size stays relatively constant, most allocated objects become garbage quickly. Figure 6 shows a theoretical profile where the live heap size grows during each phase of computation. This could (though doesn't have to) represent bloat.

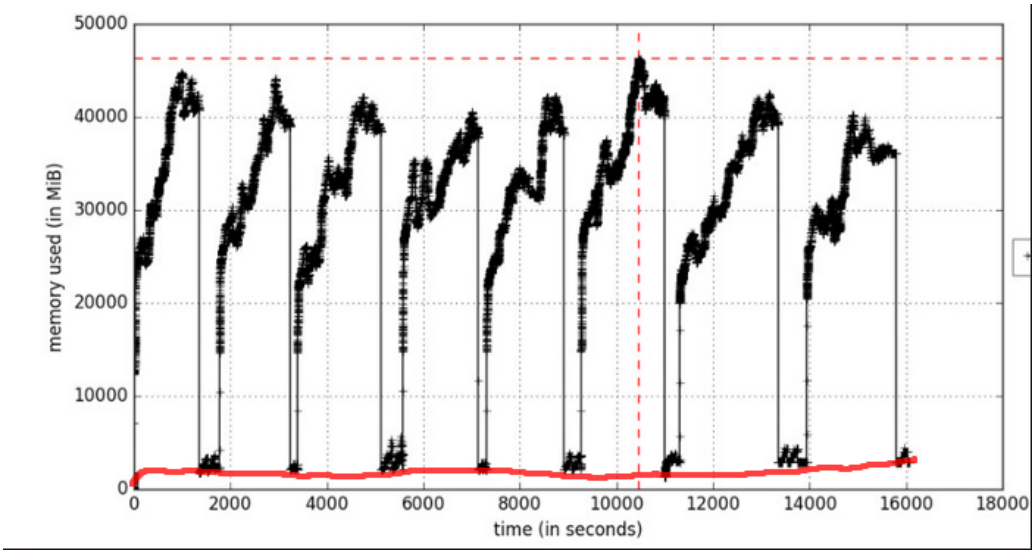


Figure 5: Live heap profile if the application maintained a low live heap size, which would be indicative of a large amount of churn.

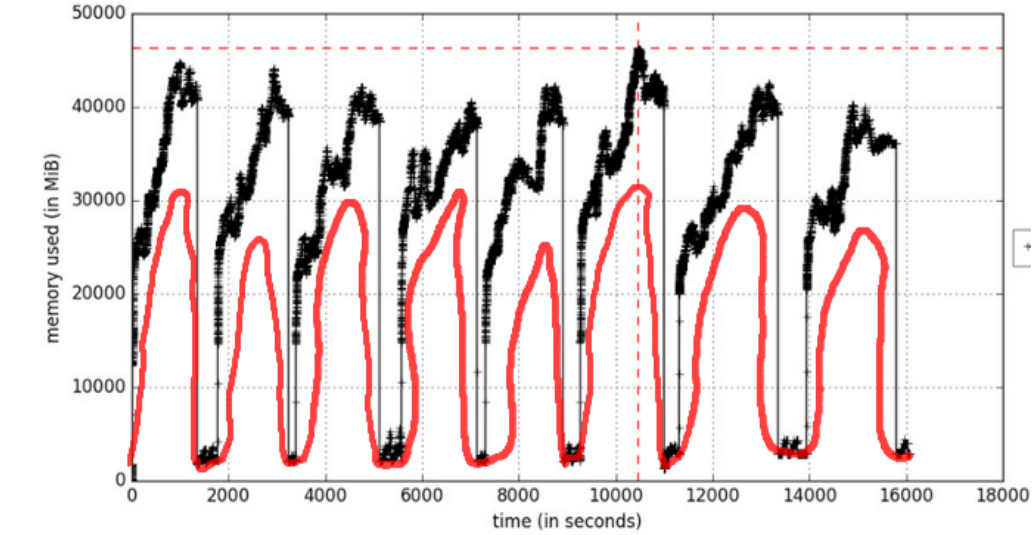


Figure 6: Live heap profile if the application's increased memory usage was largely driven by live objects, which could be due to bloat.

2. (4 pts) Based on how you answered the prior question (i.e., how you completed the chart), does the completed chart depict an application that suffers from any of bloat, churn, or a leak? Briefly justify your answer.

4 Memory Management (20 pts)

4.1 Memory-related Errors (8 pts)

Consider the following C program, which is a rephrased excerpt from a student's p3 implementation:

```
1  #define NUM_FOO 20
2  #define NUM_ARR 50
3
4  #include <string.h>
5  #include <assert.h>
6
7  struct foo {
8      char bar[NUM_ARR];
9      char baz[NUM_ARR];
10 };
11
12 static struct foo foo_arr[NUM_FOO];
13 static int num;
14
15 int main(int argc, char ** argv) {
16     num = 1024;
17
18     for (int i = 0; i <= NUM_FOO; i++) {
19        memset(foo_arr[i].bar, 'A', NUM_ARR);
20        memset(foo_arr[i].baz, 'B', NUM_ARR);
21     }
22
23     assert(num == 1024);
24 }
```

1. (2 pts) What memory-related bug does this program contain?

   Solution: It contains an array bounds violation (line 18: `i <= NUM_FOO` should be `i < NUM_FOO`)

2. (2 pts) Although memory-related bugs in general cause undefined (and thus unpredictable) behavior, in practice, such undefined behavior will manifest itself in different

ways. When compiled as follows:

```
$ gcc memerror.c -o memerror
```

the program outputs:

```
$ ./memerror
memerror: memerror.c:23: main: Assertion `num == 1024' failed.
Aborted
```

Briefly explain why the assertion fails!

Solution: The variable `num` must have been allocated at the same address in the BSS segment as `foo_arr[NUM_FOO]` and was therefore overwritten on lines 19/20.

3. (2 pts) Now consider compiling the program like so:

```
$ gcc -O2 memerror.c -o memerror
```

the assertion failure no longer occurs:

```
$ ./memerror
```

Briefly explain what might have caused the assertion to no longer fail!

Solution: The compiler can detect that the static variable `num` isn't being accessed anywhere except lines 16 and 23; it can therefore determine the outcome of the equality test at compile time. Consequently, the compiler will remove the `assert()` statement altogether.

4. (2 pts) When running under the undefined behavior sanitizer (UBSAN), the error is flagged. How does UBSAN flag this error?

Solution: UBSAN instruments the code to add out-of-bounds checks, just like they are present in Java. Therefore, it can flag when the out-of-bounds accesses on lines 19/20 occur.

```
memerror.c:19:22: runtime error: index 20 out of bounds for type 'foo [20]'
memerror.c:20:22: runtime error: index 20 out of bounds for type 'foo [20]'
```

4.2 malloc/free (12 pts)

Prof. ARB is trying to improve upon the traditional explicit free memory space management (`malloc()/free()` etc.), but **without** breaking the existing API. He is thinking of a number of ways to do so. Indicate if the following statements are **true** or **false**.

1. (2 pts) Using fixed size blocks can remove both internal and external fragmentation for some applications.

   True - if the applications' allocation request sizes are clustered

2. (2 pts) A lazy algorithm that waits for several `malloc()` requests and then allocates them in a batch can help reduce fragmentation.

   False - this would break the existing API

3. (2 pts) A hybrid method that alternatively does *best fit* and *next fit* can be achieved in O(1) overall.

   False - best fit is generally not possible in constant time

4. (2 pts) A hybrid method that alternatively does *best fit* and *next fit* would typically yield less fragmentation compared to the best fit approach alone.

   False - adding a next fit is unlikely to reduce fragmentation

5. (2 pts) Consider a scanning algorithm that examines all of the allocated blocks and packs them tightly to reduce fragmentation by moving/rearranging the blocks. If the overhead of such an algorithm can be reduced (or amortized over a large number of requests), it can serve as a promising allocator.

   False - compaction is not possible without breaking the existing API

6. (2 pts) Fragmentation tends to decrease if blocks of different sizes are immediately coalesced in a segregated list based approach.

   True - that's why we use it