CS 3214 Spring 2020 Final Exam Solutions

May 14, 2020

Contents

1 Input and Output (16 pts) **2**

2 Running Threads (12 pts) **3**

3 Dynamic Memory Management (18 pts) **4**
- 3.1 Sequence 1 (8 pts) . . . 5
- 3.2 Sequence 2 (10 pts) . . . 6

4 Automatic Memory Management (18 pts) **8**
- 4.1 Object Reachability Graph, Take 1, (12 pts) . . . 8
- 4.2 Object Reachability Graph, Take 2, (6 pts) . . . 9

5 Virtual Memory (18 pts) **10**
- 5.1 On-demand Paging (6 pts) . . . 10
- 5.2 Files and Memory (6 pts) . . . 13
- 5.3 The 3 C's of Virtual Memory (6 pts) . . . 15

6 Networking (18 pts) **16**
- 6.1 File Transfer (6 pts) . . . 16
- 6.2 JWT (6 pts) . . . 17
- 6.3 Too QUIC? (6 pts) . . . 17

7 Submission Requirements **18**

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. If you have a question, you may post it as a *private* question on Piazza.

1 Input and Output (16 pts)

As part of this class, we have the goal of ensuring that everyone has mastered basic Unix skills. Consider the shell script `runmyst.sh` shown below. It invokes a command `./myst` which refers to an executable in the current directory.

When sourced from a bash shell, you would see:

```
$ source runmyst.sh
output 1
one
two
three
output 2
four
five
```

Write a program `myst.c` that, when compiled to `myst`, will produce this exact output when it is invoked as part of running this shell script.

You may not hardwire any of the values of the variables shown at the beginning of the script (VAR1, VAR2, etc.) in your program. In other words, your program should still produce equivalent output if any of these are replaced in the shell script. Each variable is assumed to be a distinct sequence of only alphanumeric characters; that is, `"VAR1=one"` could be replaced with `"VAR1=OneOne"`, but not with `"VAR1=one one"` or `"VAR1=2>one"` or similar.

```bash
#!/bin/bash
VAR1=one
VAR2=two
VAR3=three
VAR4=four
VAR5=five
FILE=file
STDERR=stderr
STDOUT=stdout
STDIN=stdin

echo -n ${VAR1} > ${STDIN}
echo -n ${VAR4} > ${FILE}

export ENVVAR=${VAR3}
./myst < ${STDIN} ${VAR2} ${VAR5} ${FILE} 2> ${STDERR} | cat > ${STDOUT}

echo "output 1"
# must be VAR1, VAR2, VAR3 on 3 lines
cat ${STDOUT}
echo "output 2"
# must be VAR4, VAR5 on 2 lines
cat ${STDERR}
```

Solution: this program treats the third argument as the name of a file whose content is output to standard error, followed by a newline, followed by the second argument itself, followed by a newline.

To standard output, this program outputs first the content of its standard input stream, followed by a newline, followed by the first argument, followed by a newline, followed by the value of the environment variable `ENVVAR`.

Since the variable's values were guaranteed to be single words, it suffices to read a single token from standard input or the file given in `av[3]`, respectively.

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

int
main(int ac, char *av[])
{
    FILE * infile = fopen(av[3], "r");
    int c;
    while ((c = fgetc(infile)) != EOF)
        fputc(c, stderr);
    fprintf(stderr, "\n%s\n", av[2]);

    while ((c = fgetc(stdin)) != EOF)
        fputc(c, stdout);
    printf("\n%s\n", av[1]);
    printf("%s\n", getenv("ENVVAR"));
}
```

I was dismayed by how many students simply hardwired a limit on the size of either the standard input file stream or the file being read, or the length of the `argv` variables. Imposing such limits should never be your first thought - the world is riddled with error prone software caused by baking assumptions about (unknown) input data sizes into code. In addition, imposing such artificial limits here was completely unnecessary.

2 Running Threads (12 pts)

Process or threads that are in the RUNNING state consume CPU time. Write a (possibly multi-threaded) program that, when run with the bash builtin time command on an unloaded machine with at least 4 CPUs or cores, outputs this:

```
$ time ./twoeight

real        0m2.002s
user        0m7.998s
sys         0m0.004s
```

(The reported real time should be about 2s, the user time about 8s, the system time less than 0.01s. Small deviations of less than 0.2s are ok. Hint: use `gettimeofday(2)` and `timersub(3)`)

Solution: one way to approach this is to spawn 3 spinning threads, then spin for 2 seconds. The use of `gettimeofday`, which is a specially optimized system call, guarantees that the system time is very small.

```
#include <pthread.h>
#include <sys/time.h>

static void *loop(void *_) {
    for(;;);
}

int
main()
{
    struct timeval start;
    gettimeofday(&start, NULL);
    pthread_t tid;
    for (int i = 0; i < 3; i++)
        pthread_create(&tid, NULL, loop, NULL);

    for (;;) {
        struct timeval now;
        gettimeofday(&now, NULL);
        struct timeval delta;
        timersub(&now, &start, &delta);
        if (delta.tv_sec == 2)
            break;
    }
}
```

3 Dynamic Memory Management (18 pts)

Suppose a user-level implementation of `malloc()` implements the following policies:

- A single free list is kept.
- A first-fit policy is used.
- The free list is accessed in LIFO fashion (the most recently freed block appears at the head of the list).
- Block splitting makes use of the lower address portion of a block and puts the higher address portion onto the free list.
- Boundary tag headers and footers are used to enable coalescing.
- Freed blocks are immediately coalesced.
- If possible, `realloc()` always extends a block.

- If the allocator is out of memory, it will extend the heap in chunks of 8000 bytes each time. Initially, the heap is contains 0 bytes.

3.1 Sequence 1 (8 pts)

Consider the following sequence of calls:

```
void * a1 = malloc(2000);
void * a2 = malloc(1000);
void * a3 = malloc(1000);
free(a2);
realloc(a1, 2500)
free(a3);
```

Sketch the layout of the memory region managed by the allocator. For the purposes of this question, you may ignore the space taken up by boundary tags and link elements. Clearly denote used and free blocks of memory and their respective sizes. For instance, after a single call to `malloc(7000)`, a sketch of the heap when managed by an allocator that followed above mentioned policies would look like this:

```
<-------- used block (7000) ------->
<-------- free block (1000) ------->
```

where the first line denotes the block with the lowest address in memory.

Solution: Here we show the steps (which were not required to be shown individually)

First, an 8000 byte chunk is added to the free list and split:

```
<-------- used block (2000) ------->
<-------- free block (6000) ------->
```

then, the 6000 block is further split

```
<-------- used block (2000) ------->
<-------- used block (1000) ------->
<-------- free block (5000) ------->
```

and again

```
<-------- used block (2000) ------->
<-------- used block (1000) ------->
<-------- used block (1000) ------->
<-------- free block (4000) ------->
```

Freeing the second block results in

```
<-------- used block (2000) ------->
<-------- free block (1000) ------->
<-------- used block (1000) ------->
<-------- free block (4000) ------->
```

which allows the `realloc()` call to extend into the free block to

```
<-------- used block (2500) ------->
<-------- free block (500) ------->
<-------- used block (1000) ------->
<-------- free block (4000) ------->
```

Finally, the last block is freed and coalesced:

```
<-------- used block (2500) ------->
<-------- free block (5500) ------->
```

3.2 Sequence 2 (10 pts)

Consider the following sequence of calls:

```
void * a1 = malloc(1000);
void * a2 = malloc(500);
void * a3 = malloc(1000);
void * a4 = malloc(500);
void * a5 = malloc(1000);
void * a6 = malloc(500);
free(a4);
free(a6);
free(a5);
a4 = malloc(250);
free(a3);
free(a4);
```

and sketch the heap layout resulting from this sequence as well.

Solution: After first malloc() we get

```
<-------- used block (1000) ------->
<-------- free block (7000) ------->
```

then

```
<-------- used block (1000) ------->
<-------- used block (500) -------->
<-------- free block (6500) ------->
```

and

```
<-------- used block (1000) ------->
<-------- used block (500) -------->
<-------- used block (1000) ------->
<-------- free block (5500) ------->
```

and

```
<-------- used block (1000) ------->
<-------- used block (500) -------->
<-------- used block (1000) ------->
<-------- used block (500) -------->
<-------- free block (5000) ------->
```

and

```
<-------- used block (1000) ------->
<-------- used block (500) -------->
<-------- used block (1000) ------->
<-------- used block (500) -------->
<-------- used block (1000) ------->
<-------- free block (4000) ------->
```

and

```
<-------- used block (1000) ------->  a1
<-------- used block (500) -------->  a2
<-------- used block (1000) ------->  a3
<-------- used block (500) -------->  a4
<-------- used block (1000) ------->  a5
<-------- used block (500) -------->  a6
<-------- free block (3500) ------->
```

Freeing `a4` yields

```
<-------- used block (1000) ------->  a1
<-------- used block (500) -------->  a2
<-------- used block (1000) ------->  a3
<-------- free block (500) -------->
<-------- used block (1000) ------->  a5
<-------- used block (500) -------->  a6
<-------- free block (3500) ------->
```

Freeing `a6` yields

```
<-------- used block (1000) ------->  a1
<-------- used block (500) -------->  a2
<-------- used block (1000) ------->  a3
<-------- free block (500) -------->
<-------- used block (1000) ------->  a5
<-------- free block (4000) ------->
```

Freeing `a5` yields

```
<-------- used block (1000) ------->  a1
<-------- used block (500) -------->  a2
<-------- used block (1000) ------->  a3
<-------- free block (5500) ------->
```

The next `malloc(250)` yields

```
<-------- used block (1000) -------> a1
<-------- used block (500) --------> a2
<-------- used block (1000) -------> a3
<-------- used block (250) --------> a4
<-------- free block (5250) ------->
```

After freeing `a3` we get

```
<-------- used block (1000) -------> a1
<-------- used block (500) --------> a2
<-------- free block (1000) ------->
<-------- used block (250) --------> a4
<-------- free block (5250) ------->
```

and after freeing `a4` we have

```
<-------- used block (1000) -------> a1
<-------- used block (500) --------> a2
<-------- free block (6500) ------->
```

4 Automatic Memory Management (18 pts)

4.1 Object Reachability Graph, Take 1, (12 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 reachability graph produced by the execution of a small Java program.

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

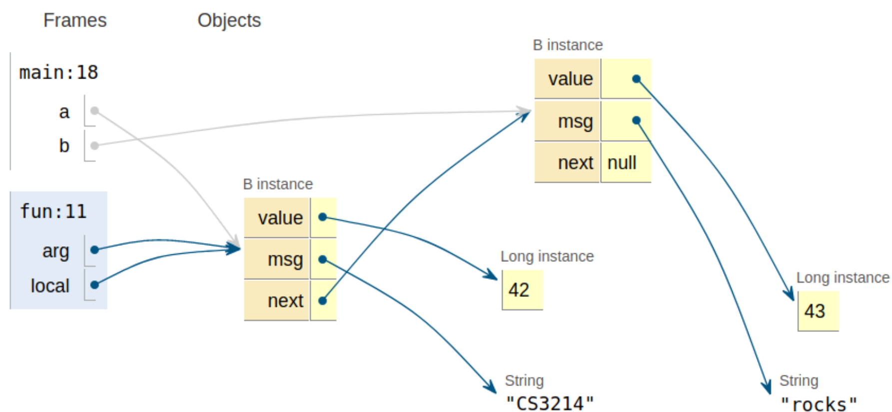

Figure 1: A snapshot of an object reachability graph produced by a Java program. On the left, roots that are part of stack frames are shown, corresponding to static methods `main` and `fun`.

Solution: A solution is:

```
public class A {
    static class B {
      Long value;
      String msg;
      B next;
    }

    static void fun(B arg) {
      B local = arg;
      local.next.value = 43l;
      local.next.msg = "rocks"; // after this line
    }

    public static void main(String[] args) {
      B a = new B();
      B b = new B();
      a.next = b;
      a.value = 42l;
      a.msg = "CS3214";
      fun(a);
    }
}
```

which can be verified using http://pythontutor.com/java.html

4.2 Object Reachability Graph, Take 2, (6 pts)

Now consider the following reachability graph.

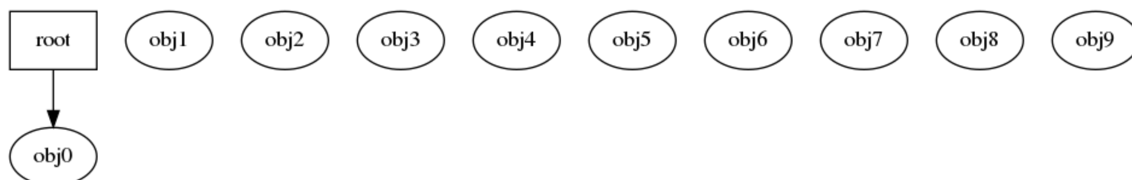

Figure 2: Object reachability graph (garbage collection has yet to occur). Roots are shown as squares, heap-allocated objects are shown as ovals.

Write a program in a garbage-collected language of your choice that would produce this reachability graph. (`root` may be an arbitrary root, and `obj0` to `obj9` may be arbitrary objects in your chosen language.)

Solution: A Java solution could be:

```
Object root;
for (int i = 0; i < 10; i++)
   root = new Object();
// snapshot now
```

which leaves 9 unreachable (garbage) objects and a single reachable object.

Python code might look like so:

```
for i in range(10):
    root = object()
# snapshot now
```

5 Virtual Memory (18 pts)

5.1 On-demand Paging (6 pts)

Like most modern OS, Linux uses fully on-demand paged virtual memory. The `getrusage` system call can be used to obtain information regarding the number of page faults a process experienced during its execution. Consider the following program `pagefaults.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <stdint.h>
#include <sys/resource.h>

static void
report_vm_stats()
{
    struct rusage u;
    getrusage(RUSAGE_SELF, &u);
    printf("ru_minflt %ld\n", u.ru_minflt);
    printf("ru_majflt %ld\n", u.ru_majflt);
}

/* you may make changes here */

int
main()
{
    /* you may make changes here */
    report_vm_stats();
}
```

When run on a Linux machine, it outputs the number of minor page faults and major page faults that have occurred from when this process was created with `fork()` to when `getrusage()` was called. The man page for `getrusage()` describes these values as:

```
ru_minflt
   The number of page faults serviced without any I/O activity; here I/O
   activity is avoided by "reclaiming" a page frame from the list of pages
   awaiting reallocation.

ru_majflt
   The number of page faults serviced that required I/O activity.
```

When compiled and run as given, this program will output the number of minor page faults it experienced (on our current rlogin machines, this number can vary slightly between runs and is approx. 175). The number differs between Linux OS versions and possibly also between the versions of the toolchains used to build the executable.

Task 1: Modify pagefaults.c such that the number of minor page faults reported increases by approximately 256. A range from 240 to 270 is acceptable. For instance, on our rlogin machines, a range from $175 + 240 = 415$ to $175 + 270 = 445$ would be acceptable. Add an explanation of what causes these additional approximate 256 page faults.

Solution: A minimal solution would allocate and touch 256 pages (assuming a page size of 4 KB), like so:

```c
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <stdint.h>
#include <sys/resource.h>

static void
report_vm_stats()
{
    struct rusage u;
    getrusage(RUSAGE_SELF, &u);
    printf("ru_minflt %ld\n", u.ru_minflt);
    printf("ru_majflt %ld\n", u.ru_majflt);
}

int
main()
{
    char * p = malloc(4096*256);
    for (char * q = p; q < p + 4096*256; q += 4096)
        *q = 0;

    report_vm_stats();
}
```

Note that making use of global variables, or even stack space, works as well, as in

```
char p[4096*256];

int
main()
{
    for (char * q = p; q < p + 4096*256; q += 4096)
        *q = 0;

    report_vm_stats();
}
```

or

```
int
main()
{
    char p[4096*256];

    for (char * q = p; q < p + 4096*256; q += 4096)
        *q = 0;

    report_vm_stats();
}
```

If you presented solutions that allocate more than 1 MB, you would need to explain how the number of minor page faults comes out to about 256.

A number of solutions tried to call `malloc(4096)` (or some other number) about 256 times, without accessing the memory that is returned by `malloc()`. If so, there would need to be an explanation of why this would cause apprx. 256 minor faults. If the client program doesn't access the memory, the only accesses would be inside the user-level memory allocator. This allocator would access only the boundary tag headers and footers to update them; for a $4,096$ allocation, once you take the space for these headers into account, you end up accessing 2 pages since the boundary tag headers are more than 1 page apart. The overall number of page faults is still in the expected range, presumably because all 256 blocks (each taking up $4,096 + x$ for some small x) lie in a roughly 1MB area. If you didn't explain this, I assume that you somehow thought that calling `malloc()` eagerly faults in memory, which is does not in a fully on-demand paged system. (If any page faults do occur, they do only as a side-effect of the user-level allocator's updating of the headers/footers.) Some solution attempts did something, like forking processes or opening files. All of these activities may cause minor page faults because they allocate memory and touch it, but there's no clear relationship between their number and the expected 256 minor faults.

Calling `calloc()` of about 1MB also causes this number of pagefaults because `calloc()` accesses the memory - it zeros it out.

A number of you allocated excessive amount of virtual memory. If you allocate such large chunks, the OS will use "huge pages" of 2MB each when an address is accessed, which reduces the number of page faults to one per 2MB range.

Task 2: We claim that modifying pagefaults.c to reliably cause a similar number of major page faults is much more difficult.

Either modify pagefaults.c to reliably cause it to encounter a similar number of major page faults, or explain why it is difficult for a user program to influence the number of major page faults they encounter during execution.

Solution: It is, in general, difficult to guarantee major page faults that require I/O activity because it is difficult to predict what the OS has cached in memory. Any file content that is cached in memory will not require I/O activity, so won't become a major page fault. In particular, data that you have just written to a file is typically still cached. User programs generally do not control what's being cached by the OS.

A second approach might be to force eviction by increasing the working set size. Then subsequent accesses would cause a major page fault. This would also be very difficult to make reliable since user programs have little control over eviction policies. Those are 2 good arguments in favor.

Note that simply stating that it would "require disk I/O" is not an argument. User programs regularly require disk I/O. Similarly, simply appealing to size is also not a precise enough argument.

Finally, a number of you discovered the `posix_fadvise` API that allows a program to "hint" to the OS that it won't be accessing a file's content. If used properly, this can indeed cause major page faults if the file data is first purged from the Linux's kernel's page cache and then mmap()'d and accessed. This is still tricky to get correct because the `POSIX_FADV_DONTNEED` flag purges the data only when called; the next access appears to reestablish the data in the cache, which means you have to call it again before the next access, or repeating the open/mmap sequence with the same or a different file. Please note that even though this exam was Open Internet, proper attribution (link) is still required.

5.2 Files and Memory (6 pts)

Unix provides facilities that can make a file's content appear as random access memory. Complete the program below by adding any missing statement(s) in `main()`!

```c
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <fcntl.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>

#define N   1000
struct pair {
    uint32_t sum;
    uint32_t fd;
};

static struct pair
```

```c
writefile(int n) {
    const char * fname = ".myfile";
    int myfd = open(fname, O_CREAT | O_TRUNC | O_RDWR, 0600);
    unlink(fname);
    uint32_t sum = 0;
    for (uint32_t i = 0; i < n; i++) {
        uint32_t rnd = rand() % n;
        sum += rnd;
        write(myfd, &rnd, sizeof (rnd));
    }

    return (struct pair){ .sum = sum, .fd = myfd };
}

int
main()
{
    srand(time(NULL));
    struct pair sumfd = writefile(N);

    /* Insert missing code here so that this program prints "Ok".
     * You may not use the read, readv, preadv, or preadv2 system calls
     */

    uint32_t sum = 0;
    for (uint32_t i = 0; i < N; i++)
        sum += addr[i];

    if (sum != sumfd.sum)
        abort();
    printf("Ok\n");
}
```

Solution: To complete the program, a call to `mmap()` could be added:

```c
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <fcntl.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
```

```c
#define N   1000
struct pair {
    uint32_t sum;
    uint32_t fd;
};

static struct pair
writefile(int n) {
    const char * fname = ".myfile";
    int myfd = open(fname, O_CREAT | O_TRUNC | O_RDWR, 0600);
    unlink(fname);
    uint32_t sum = 0;
    for (uint32_t i = 0; i < n; i++) {
        uint32_t rnd = rand() % n;
        sum += rnd;
        write(myfd, &rnd, sizeof (rnd));
    }

    return (struct pair){ .sum = sum, .fd = myfd };
}

int
main()
{
    srand(time(NULL));
    struct pair sumfd = writefile(N);

    uint32_t *addr = mmap(NULL, N*sizeof(uint32_t), PROT_READ, MAP_PRIVATE, sumfd.fd, 0);

    uint32_t sum = 0;
    for (uint32_t i = 0; i < N; i++)
        sum += addr[i];

    if (sum != sumfd.sum)
        abort();
    printf("Ok\n");
}
```

5.3 The 3 C's of Virtual Memory (6 pts)

Virtual memory can be viewed as a cache for disks or other forms of secondary storage. Data is cached in pages in RAM that may otherwise reside on disk or may be swapped out to disk.

From Computer Architecture, you are familiar with the 3 C's of cache misses: compulsory (cold), conflict, and capacity misses. These are ordinarily defined for processor caches such as the L1 cache. Let us extend these definitions to virtual memory. For each type of miss, explain whether and how it can occur in the context of virtual memory, and why.

Solution:

Compulsory misses occur in an demand-paged system when a page is first touched - for instance, when malloc'd memory is first touched, a global variable is set or read, or when even parts of the code are first invoked.

Conflict misses do not occur - it's a fully associative cache. Any page in RAM can hold any data that would otherwise be on disk.

Capacity misses occur when pages are accessed that were previously evicted to disk.

6 Networking (18 pts)

6.1 File Transfer (6 pts)

Consider the following hypothetical file transfer protocol proposal, which consists of multiple steps:

- A client opens a TCP connection to a server.
- Upon successful connection, the client sends `GET <filename>` followed by a CRLF to the sender, where `<filename>` is the name of the file the client wants to retrieve.
- The server retrieves the file from its file system and responds with the file's content.
- The client saves the content to a local file as it receives it.
- After the file is sent, the server closes the connection by calling `close()` on the file descriptor of the TCP connection.
- If the client learns that the connection has been closed by the server, it will exit after having saved the content to a local file.

Answer the following 2 questions regarding the feasibility of this proposed protocol:

a) If the protocol were implemented as described, would it be possible for the client to learn when the server completes the file transfer?

b) Based on your knowledge of transport layer protocols, would this hypothetical protocol guarantee the reliable and complete transfer of files from the server to the requesting client?

Solution: The answers to both question is yes.

The client learns when the server completes the file transfer because it encounters an EOF condition on the TCP connection (e.g., a call to `read()` or `recv()` returns zero bytes and no error).

TCP provides reliable connections (including establishment and teardown), so the client can be sure the entire file has been transferred when it sees the connection has been closed (without encountering any errors). As an aside, no Content-Length: or similar header is necessary which this protocol does not include.

In fact, above file transfer protocol is near identical to Tim Berner-Lee's first version of the HTTP protocol (now referred to as "version 0.9") that started the WWW.

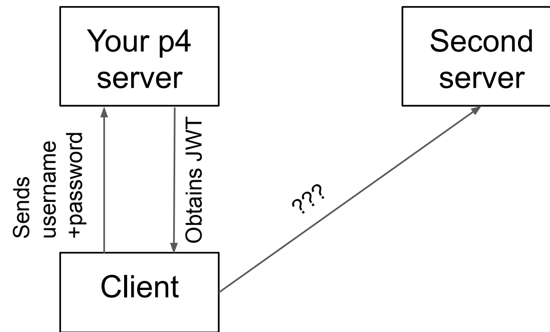

Figure 3: Could a client send the JWT obtained from your p4 to another server?

6.2 JWT (6 pts)

Consider your use of JWT in project 4, and consider the scenario shown in Figure 3.

How would you change your p4 implementation so that users can authenticate with your server, but access the "second server" after successful authentication? You should assume that this server is operated by an organization that the client trusts, but that is not affiliated with the operator of your modified p4 server.

Describe what changes you would need to make. If your implementation already does part of what would be required to accommodate this scenario, state it.

Solution: JWT represent cryptographically signed claims and are well suited for this scenario. In fact, they are widely used in applications that use separate identity providers. The identity provider (your p4) server would issue a JWT token that the client then presents to the second server. The second server would then verify that the JWT was signed by your server.

With the assumption that the second server belongs to a different organization, it would not make sense to share a secret key with it. Therefore, RS256 should be used. Your server would then publish its public key to the second server.

(Sharing the secret key as in HS256 would allow the second server to create JWT tokens for any users without having access to the password database.)

6.3 Too QUIC? (6 pts)

The QUIC Protocol is being hailed as a key component of HTTP/3. A Google search as of the time of this writing yields the following panel:

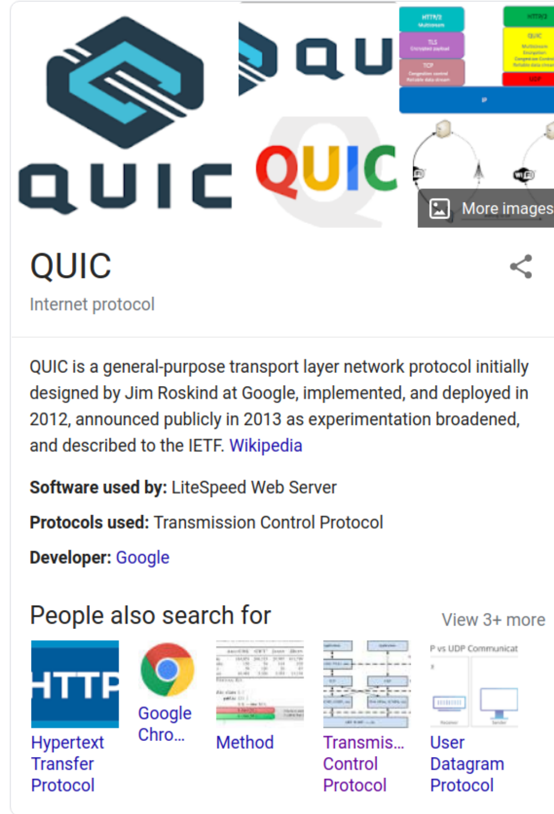

Figure 4: Google Search Result Panel for QUIC

Which important detail did Google's AI (which apparently compiled this panel) get wrong about QUIC?

Solution: QUIC doesn't actually use TCP, it replaces it with its own reliable transport protocol. That's one of its key innovations that allows it to overcome transport layer blocking.

7 Submission Requirements

Submit a tar file that contains the following files:

- `myst.c` to answer Question 1
- `twoeight.c` to answer Question 2
- `malloc.txt` with answers to both parts of Question 3
- `A.java` to answer Question 4.1

- `reachability.txt` to answer Question 4.2. This will contain source code in your chosen language, but for uniformity, please give it this name.
- `pagefault.c` to answer Question 5.1, Task 1 and 2. Please answer Task 2 using a comment (or code separated with an if statement) in this file.
- `randomsum.c` to answer Question 5.2
- `vmcache.txt` to answer Question 5.3 (3 answers required)
- `networking.txt` with answers to Question 6. This means 4 separate answers for 6.1 (a), (b), 6.2, and 6.3.