Exam 4 Cover Sheet

I affirm that I have completed this exam without unauthorized assistance from any person, materials, or device. I also affirm that I have completed the exam according to the restrictions listed in the exam document.

Signed: ______________________________

Print name: ______________________________

Date: ______________________________

Please include a signed and dated copy of this cover sheet at the beginning of your exam submission.

You will not receive credit for the exam unless your exam includes this (signed and dated) cover sheet.

CSF Fall 2020, Exam 4

Due: Wednesday, December 16th by 5:00pm EDT

The permitted resources for this exam are:

- The textbook(s)
- Materials posted directly on the course website (e.g., slides)
- Materials linked directly from the course website (e.g., assembly language resources)

Do *not* use any resources other than the ones explicitly noted above.

You may *not* write program(s) or use automated calculation devices or programs. You will need to do required calculations by hand. In other words, this is a "pencil and paper" exam, but you may type your answers. (It is also fine to hand-write your answers, just make sure they're legible.)

Do *not* discuss the exam with anyone else: your answers must be your own answers.

You will submit your answers to Gradescope as "Exam 4" in PDF format. When you upload to Gradescope, you will need to select the page of your submitted document corresponding to your answer to each question. You may use software (word processor, LaTeX, etc.) to prepare your answers. Make sure that your submission includes the cover sheet (signed and dated) as its first page.

Important: Show your work, and justify your answers. "Bare" answers (without supporting work or justification) may not receive full credit.

Powers of 2 ($2^y = x$):

y	0	1	2	3	4	5	6	7	8	9	10	11	12
x	1	2	4	8	16	32	64	128	256	512	1,024	2,048	4,096

y	13	14	15
x	8,192	16,384	32,768

[Questions begin on next page]

Question 1. [15 points] Assume that two processes, A and B, are communicating using a TCP socket, and that each process uses an `int` variable called `sockfd` to refer to the socket at its end of the TCP connection. Also, assume that process A executes the code

```
uint32_t val = 0xb977514a;
rio_writen(sockfd, &val, 4);
```

and that process B executes the code

```
uint8_t buf[4];
rio_readn(sockfd, buf, 4);
printf("%02x%02x%02x%02x\n", buf[0], buf[1], buf[2], buf[3]);
```

Note that `rio_writen` and `rio_readn` work the same way as `read` and `write`, except that they loop until all of the data is written or read.

Assuming that no errors occur sending or receiving the data, what possible output(s) will be printed by process B? Explain briefly.

Question 2. [30 points] Consider the following `struct` data type and declarations for the `sr_send` and `sr_receive` functions:

```
struct SensorReading {
  float pos_x, pos_y;
  unsigned event_count;
};

int sr_send(int fd, const struct SensorReading *sr);
int sr_receive(int fd, struct SensorReading *sr);
```

The `sr_send` function should send the data in the given `struct SensorReading` instance via the specified file descriptor (`fd`). The `sr_receive` function should read the data sent by `sr_send` and store it in the specified `struct SensorReading` instance. Both functions should return 1 if successful, 0 otherwise.

Example use of `sr_send`:

```
struct SensorReading sr =
  get_sensor_reading();
sr_send(sockfd, &sr);
```

Example use of `sr_receive`:

```
struct SensorReading sr;
sr_receive(sockfd, &sr);
```

Show implementations of `sr_send` and `sr_receive`. The `sr_receive` function may assume that the first byte of data it reads is the first byte sent by `sr_send`. **Important**: `sr_receive` should not read any data that is past the last byte sent by `sr_send`. The choice of data format is up to you. You may use functions in `csapp.h` and `csapp.c`. You may use C or C++, and your implementation code may use standard C and C++ library functions.

Question 3. [10 points] Assume that `prog1` and `prog2` are programs which read input from standard input and write output to standard output. Consider the following ways of running these programs to transform an input file `input.txt` into an output file `output.txt`:

Option 1:

```
./prog1 < input.txt > tmp.txt
./prog2 < tmp.txt > output.txt
rm tmp.txt
```

Option 2:

```
./prog1 < input.txt | ./prog2 > output.txt
```

(a) [7 points] Explain why Option 2 might complete faster than Option 1 on a system with a multi-core CPU.

(b) [3 points] Is it possible that Option 2 might be faster to complete than Option 1 even on a single-core CPU? Briefly explain why or why not.

Hint: The `|` character in a Unix shell command creates a pipe feeding the standard output of one process to the standard input of another.

Question 4. [10 points] Consider the following main loop implementing a network server application using processes for concurrency:

```
int server_fd = Open_listenfd(port);

while (1) {
  int client_fd = Accept(server_fd, NULL, NULL);
  pid_t pid = Fork();
  if (pid == 0) {
    // in child
    chat_with_client(client_fd);
  }
  close(client_fd);
}
```

Assume that

- `Open_listenfd`, `Accept`, and `Fork` are functions from `csapp.h` and `csapp.c`
- the `chat_with_client` function correctly handles all details of communicating with the remote client, including closing the client socket file descriptor
- the main server process has a handler for `SIGCHLD` to wait for child processes to exit
- the server intentionally does not place any limit on how many client sub-processes can be running simultaneously
- the server intentionally does not have any mechanism for handling shutdown requests

Describe the most significant bug in this main loop, and how to fix it.

Question 5. [30 points] Consider the following `struct Histogram` data type and functions:

```
struct Histogram {
  unsigned num_buckets;
  unsigned *count_array;
};

struct Histogram *hist_create(unsigned num_buckets) {
  struct Histogram *h = malloc(sizeof(struct Histogram));
  h->num_buckets = num_buckets;
  h->count_array = calloc(num_buckets, sizeof(unsigned));
  return h;
}

void hist_increment(struct Histogram *h, unsigned bucket) {
  assert(bucket < h->num_buckets);
  h->count_array[bucket]++;
}

unsigned hist_get_count(struct Histogram *h, unsigned bucket) {
  assert(bucket < h->num_buckets);
  return h->count_array[bucket];
}
```

This purpose of the data structure is to maintain an array of count values, one per "bucket". A "bucket" is an index into the array of count values. The `hist_increment` function increases the count in a specified bucket by 1, and the `hist_get_count` function allows the caller to get the current count value of a specified bucket.

Show how to modify this data type so that it may be safely used by multiple threads. Your modification should ensure that the count values accurately reflect all of the calls to `hist_increment` that have been made by all threads.

Question 6. [5 points] Briefly describe a way that the `struct Histogram` data type from Question 5 could allow calls to `hist_increment` incrementing different buckets to proceed in parallel, while still guaranteeing that the counts in the array will be updated correctly (i.e., lost updates are not possible.) You don't need to show a complete implementation, but you should describe how to make this work in a reasonable amount of detail.