Exam 4 Fall 2020 Solution

1) The output printed by process B depends on the byte order of the system executing process A. If process A is on a little-endian architecture, then multi-byte values are arranged from least significant to most significant, process B's output would be 4a5177b9. If process A is on a big-endian architecture, where the most significant byte is first, process B's output would be b977514a.

2) Implementations using text-based format, one value per line (this is kind of complicated, since calling rio_readlineb in sr_receive isn't really an option):

```
int sr_send(int fd, const struct SensorReading *sr) {
  char buf[256];
  int len;
  len = sprintf(buf, "%59f\n", sr->pos_x);
  assert(len == 60);
  if (rio_writen(fd, buf, len) != len) { return 0; }
  len = sprintf(buf, "%59f\n", sr->pos_y);
  assert(len == 60);
  if (rio_writen(fd, buf, len) != len) { return 0; }
  len = sprintf(buf, "%10u\n", sr->event_count);
  assert(len == 11);
  if (rio_writen(fd, buf, len) != len) { return 0; }
  return 1;
}

int sr_receive(int fd, struct SensorReading *sr) {
  char buf[256];
  if (rio_readn(rd, buf, 60) != 60) { return 0; }
  sscanf(buf, "%f", &sr->pos_x);
  if (rio_readn(rd, buf, 60) != 60) { return 0; }
  sscanf(buf, "%f", &sr->pos_y);
  if (rio_readn(rd, buf, 11) != 11) { return 0; }
  sscanf(buf, "%u", &sr->event_count);
  return 1;
}
```

Implementations writing and reading raw binary data (allowed per clarifications on Piazza, and this will only work correctly on machines sharing the same byte-level representation for float and unsigned):

```
int sr_send(int fd, const struct SensorReading *sr) {
  if (rio_writen(fd, &sr->pos_x, sizeof(float)) != sizeof(float))
    { return 0; }
```

```
  if (rio_writen(fd, &sr->pos_y, sizeof(float)) != sizeof(float))
    { return 0; }
  if (rio_writen(fd, &sr->event_count, sizeof(unsigned)) != sizeof(unsigned))
    { return 0; }
  return 1;
}

int sr_receive(int fd, struct SensorReading *sr) {
  if (rio_readn(fd, &sr->pos_x, sizeof(float)) != sizeof(float))
    { return 0; }
  if (rio_readn(fd, &sr->pos_y, sizeof(float)) != sizeof(float))
    { return 0; }
  if (rio_readn(fd, &sr->event_count, sizeof(unsigned)) != sizeof(unsigned))
    { return 0; }
  return 1;
}
```

3)

(a) On a multi-core CPU, Option 2 allows prog1 and prog2 can execute in parallel. As long as prog1 sends data to prog2 in a relatively continuous stream, prog2 can do useful work while prog1 is executing, leading to a faster overall execution time, since in Option 1, prog2 doesn't even begin executing until prog1 has completed.

(b) Even on a single-core CPU, in Option 2, if either process is suspended waiting for I/O to complete (which is very possible since prog1 is reading data from a file and prog2 is writing data to a file), the other process could still execute and do useful work.

4) The problem is that the child process doesn't exit, meaning that it will attempt to call Accept after chat_with_client returns. To fix:

```
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);
    exit(0); // ←add this line
  }
  close(client_fd);
}
```

5) Using a single mutex to synchronize all accesses:

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

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));
  pthread_mutex_init(&h->lock, NULL);
  return h;
}

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

unsigned hist_get_count(struct Histogram *h, unsigned bucket) {
  assert(bucket < h->num_buckets);
  pthread_mutex_lock(&h->lock);
  unsigned count = h->count_array[bucket];
  pthread_mutex_unlock(&h->lock);
  return count;
}
```

6) A really easy solution would be to avoid using a mutex at all, and add the `_Atomic` type qualifier to the array element type:

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

This change would have the effect of making the compiler emit an atomic increment instruction in the `hist_increment` function.

Another approach would be to use an array of mutexes, one per bucket. Calls to `hist_increment` and `hist_get_count` would only need to lock and unlock the mutex for the specific bucket being accessed. The `hist_create` function would need to use `pthread_mutex_init` to initialize each member of the mutex array.