

```

1 // GROUP B
2 // Nathan Baker
3 // nathan.t.baker@okstate.edu
4
5 #include "header.h"
6 #define SUMMARY 0 // for semaphore referencing
7
8 pthread_mutex_t lock;
9
10 int get_customer_info(int socket, struct clientInformation* c) {
11     // repeatedly send prompts and scan customer responses.
12     // fill clientInformation structure.
13     char m[1000];
14     strcpy(m, "0Please enter your full name: ");
15     send(socket, &m, sizeof(m), MSG_NOSIGNAL);
16     read(socket, &m, sizeof(m));
17     sscanf(m, "%50[^\n]", c->ClientName);
18     printf("%s\n", c->ClientName);
19     strcpy(m, "0Please enter your date of birth [MM/DD/YYYY]: ");
20     send(socket, &m, sizeof(m), MSG_NOSIGNAL);
21     read(socket, &m, sizeof(m));
22     sscanf(m, "%50[^\n]", c->DateOfBirth);
23     printf("%s\n", c->DateOfBirth);
24     strcpy(m, "0Please enter your gender [M, F, Other]: ");
25     send(socket, &m, sizeof(m), MSG_NOSIGNAL);
26     read(socket, &m, sizeof(m));
27     sscanf(m, "%10[^\n]", c->Gender);
28     printf("%s\n", c->Gender);
29     strcpy(m, "0Please enter your GovernmentID number: ");
30     send(socket, &m, sizeof(m), MSG_NOSIGNAL);
31     read(socket, &m, sizeof(m));
32     sscanf(m, "%d", &c->GovernmentID);
33     printf("%d\n", c->GovernmentID);
34     strcpy(m, "0Please enter your desired date of travel [MM/DD/YYYY]: ");
35     send(socket, &m, sizeof(m), MSG_NOSIGNAL);
36     read(socket, &m, sizeof(m));
37     sscanf(m, "%50[^\n]", c->DateOfTravel);
38     printf("%s\n", c->DateOfTravel);
39     strcpy(m, "0Please enter the number of travelers: ");
40     send(socket, &m, sizeof(m), MSG_NOSIGNAL);
41     read(socket, &m, sizeof(m));
42     sscanf(m, "%d", &c->NumberOfTravelers);
43     printf("%d\n", c->NumberOfTravelers);
44     return 0;
45 }
46
47 int get_customer_ticket(int socket, struct clientInformation* c) {
48     // ask customer for their ticket number and scan the response into the struct.
49     char m[1000];
50     int ticket;
51     strcpy(m, "0Please enter your ticket number: ");
52     send(socket, &m, sizeof(m), MSG_NOSIGNAL);
53     read(socket, &m, sizeof(m));
54     sscanf(m, "%d", &c->ticket);
55     printf("%d\n", c->ticket);
56     return 0;
57 }
58
59 int change_read_count(int offset) {
60     // file used to share readcount variable between servers.
61     // update readcount by offset.
62     FILE *fp;
63     fp = fopen("summary_read_count.txt", "r");
64     int num;
65     if (fp == NULL) num = 0;
66     else {
67         fscanf(fp, "%d", &num);
68         fclose(fp);
69     }
70     if (offset == 0) return num;
71     fp = fopen("summary_read_count.txt", "w");
72     fprintf(fp, "%d", num+offset);
73     fclose(fp);
74     return num+offset; // return new readcount
75 }
76
77 int verify_enough_seats(int socket, int train, struct clientInformation* c) {
78     // compare seats in train file to seats requested.
79     // at this point thread already has access to train semaphore.
80     int available = seatChecker(train);
81     if ((c->NumberOfTravelers) > available) { // if not enough seats
82         char m[1000];
83         snprintf(m, 1000, "1Sorry, there are only %d seats available for the selected date.\nReservation cancelled.\n", available);
84         send(socket, &m, sizeof(m), MSG_NOSIGNAL);
85         signal_write(train);
86         return -1; // send customer back to menu.
87     }
88     return 0;
89 }
90
91 int confirm_purchase(int socket, int train, struct clientInformation* c) {
92     // ask user for confirmation.
93     char m[1000];
94     snprintf(m, 1000, "0\nDo you want to make reservation (yes/no): ");
95     send(socket, &m, sizeof(m), MSG_NOSIGNAL);
96     read(socket, &m, sizeof(m));
97     if (strcmp(m, "yes") == 0) return 0; // proceed.
98     else {
99         snprintf(m, 1000, "1Reservation cancelled.\n");
100         send(socket, &m, sizeof(m), MSG_NOSIGNAL);
101         signal_write(train); // release semaphore.
102         return -1; // send customer back to menu.
103     }
104 }
105
106 int confirm_cancel(int socket, struct clientInformation* c) {
107     // ask user for confirmation.
108     char m[1000];

```

```

109     sprintf(m,1000,"0\nAre you sure you want to cancel your reservation (yes/no: ");
110     send(socket, &m, sizeof(m), MSG_NOSIGNAL);
111     read(socket, &m, sizeof(m));
112     if (strcmp(m,"yes") == 0) return 0; // proceed.
113     else {
114         return -1; // send customer back to menu.
115     }
116 }
117
118 int confirm_modify(int socket, struct clientInformation* c) {
119     // inform user about modification constraints and ask for confirmation.
120     char m[1000];
121     sprintf(m,1000,"0\nReservation modifications include reducing the number of seats or changing seat choice.\nIf you want to reserve additional seats you must make a new reservation.\nAre you sure you
122     send(socket, &m, sizeof(m), MSG_NOSIGNAL);
123     read(socket, &m, sizeof(m));
124     if (strcmp(m,"yes") == 0) {
125         int n;
126         sprintf(m,1000,"0What is your new desired number of travelers [up to %d]? ",c->NumberOfTravelers);
127         send(socket, &m, sizeof(m), MSG_NOSIGNAL);
128         read(socket, &m, sizeof(m));
129         sscanf(m,"%d",&n); // read customer response
130         if (n > c->NumberOfTravelers || n < 1) { // ensure that customer requested a valid number of seats.
131             sprintf(m,1000,"1Invalid selection. Modification cancelled.\n");
132             send(socket, &m, sizeof(m), MSG_NOSIGNAL);
133             read(socket, &m, sizeof(m));
134             return -1; // send customer back to menu.
135         }
136         return n; // return the new NumberOfTravelers
137     } else {
138         return -1; // send customer back to menu.
139     }
140 }
141
142 void send_available_seats(int socket, int train, struct clientInformation* c) {
143     char output[100];
144     show_available(train, output); // populates output with train string.
145     char m[1000];
146     sprintf(m,1000,"0\nPlease choose %d of the following available seats [only spaces between each seat]:\n%s\n",c->NumberOfTravelers,output);
147     send(socket, &m, sizeof(m), MSG_NOSIGNAL); // send message asking custoemr to pick from available seats.
148 }
149
150 void show_available(int trainNum, char* output) {
151     FILE *fp ;
152     char c;
153
154     printf("Opening the file train in read mode \n");
155     if (trainNum == 1) {
156         fp = fopen ("train1.txt","r"); // opening an existing file
157     } else if (trainNum == 2) {
158         fp = fopen ("train2.txt", "r"); // opening an existing file
159     }
160     if (fp == NULL) {
161         printf ("Could not open file train \n");
162         return;
163     }
164     printf("Reading train file.\n");
165     int count = 0;
166     int char_index = 0;
167     while (1) {
168         c = fgetc (fp); // read one character = one seat
169         if (c == '0') { // available
170             output[char_index++] = 'A'+(count / 5);
171             output[char_index++] = '0'+(count % 5 + 1); // create seat number via pointer arithmetic.
172         } else if (c == '1') { // unavailable
173             output[char_index++] = '-';
174             output[char_index++] = '-';
175         } else {
176             output[char_index++] = '\0'; // end
177             break;
178         }
179         output[char_index++] = ' ';
180         count++;
181         if ((count % 5) == 0) output[char_index++] = '\n'; // new row
182     }
183     printf("Closing the file train \n");
184     fclose (fp); // Closing the file
185     return;
186 }
187
188 int check_seat(int train, int row, int column) {
189     // check if particular seat in train is available.
190     // thread already has semaphore.
191     if (!(row < 5 && row >= 0 && column < 5 && column >= 0)) return -1; // if invalid seat.
192     FILE *fp;
193     if (train == 1) {
194         fp = fopen ("train1.txt", "r");
195     } else if (train == 2) {
196         fp = fopen ("train2.txt", "r");
197     }
198     int index = row*5 + column; // 2d -> 1d address
199     printf("%d, %d, %d\n",row, column, index);
200     char c;
201     for (int i=0; i<index; i++) { // loop until seat in question.
202         c = fgetc(fp);
203     }
204     c = fgetc(fp);
205     fclose(fp);
206     if (c == '0') return 0; // available
207     else return -1; // unavailable
208 }
209
210 int write_seat(int train, int row, int column, int update) {
211     // updates individual seat status in train file.
212     // thread already holds semaphore.
213     FILE *fp;
214     if (train == 1) {
215         fp = fopen ("train1.txt", "r+");
216     } else if (train == 2) {
217         fp = fopen ("train2.txt", "r+");

```

```

218 }
219 int index = row*5 + column; // 2d -> 1d address.
220 fseek(fp, index, SEEK_SET); // go to character index.
221 if (update == 1) fputc('1',fp);
222 else if (update == 0) fputc('0',fp); // write new value.
223 fclose(fp);
224 return 0;
225 }
226
227 int verify_selection(int socket, int train, struct clientInformation* c, char* m) {
228     // thread already holds semaphore.
229     char n[1000];
230     strcpy(n,m); // make copy of seat selection string to allow memmove without losing original data.
231     printf("%s\n",n);
232     char seat[3];
233     int offset; // to hold number of scanned bytes
234     for (int i=0; i<c->NumberOfTravelers; i++) {
235         if (sscanf(n, " %2c%n", seat,&offset) != 1) { // could not scan two non-space characters
236             printf("seat verification failed.\n");
237             signal_write(train);
238             char msg[1000];
239             strcpy(msg, "1\nError: not enough seats were selected. Reservation cancelled.\n");
240             send(socket, &msg, sizeof(msg), MSG_NOSIGNAL);
241             return -1; // send customer back to menu.
242         }
243         memmove(n, n+offset, 1000); // move string head pointer by number of bytes read.
244         int row = seat[0] - 65; // 'A' -> 0
245         int column = seat[1] - 49; // '1' -> 0
246         if (check_seat(train,row,column) == -1) { // seat not available.
247             printf("seat verification failed.\n");
248             signal_write(train);
249             char msg[1000];
250             strcpy(msg, "1\nError: one or more of the selected seats is not available. Reservation cancelled.\n");
251             send(socket, &msg, sizeof(msg), MSG_NOSIGNAL);
252             return -1; // send customer back to menu.
253         }
254     }
255     if (sscanf(n, " %2c%n", seat,&offset) == 1) { // extra seat was requested.
256         printf("seat verification failed.\n");
257         signal_write(train);
258         char msg[1000];
259         strcpy(msg, "1\nError: too many seats were selected. Reservation cancelled.\n");
260         send(socket, &msg, sizeof(msg), MSG_NOSIGNAL);
261         return -1; // send customer back to menu.
262     }
263     strcpy(c->seats, m); // fill struct field with seat selection.
264     return 0;
265 }
266
267 int add_to_train(int train, struct clientInformation* c, char* m) {
268     // reserve seats in train file.
269     // thread already holds semaphore.
270     char output[100];
271     show_available(train, output); // for server-side output only
272     printf("%s\n",output);
273     char n[1000];
274     strcpy(n,m); // to move string pointer without losing original data.
275     char seat[3];
276     int offset;
277     for (int i=0; i<c->NumberOfTravelers; i++) { // loop through selected seats..
278         sscanf(n, " %2c%n", seat,&offset); // scan seat number.
279         memmove(n, n+offset, 1000); // move forward by number of scanned bytes.
280         int row = seat[0] - 65; // 'A' -> 0
281         int column = seat[1] - 49; // '1' -> 0
282         write_seat(train,row,column,1); // set seat to unavailable
283     }
284     show_available(train, output); // for server-side output only
285     printf("%s\n",output);
286     return 0;
287 }
288
289 int remove_from_train(struct clientInformation* c, int train) {
290     // open seats in train file.
291     // thread already holds semaphore.
292     char output[100];
293     show_available(train, output); // for server-side output only
294     printf("%s\n",output);
295     char n[100];
296     strcpy(n,c->seats);
297     char seat[3];
298     int offset;
299     printf("num: %d\n seats: %s\n",c->NumberOfTravelers,n);
300     for (int i=0; i<c->NumberOfTravelers; i++) {
301         sscanf(n, " %2c%n", seat,&offset); // scan seat number.
302         memmove(n, n+offset, 100); // move forward by number of scanned bytes.
303         int row = seat[0] - 65; // 'A' -> 0
304         int column = seat[1] - 49; // '1' -> 0
305         write_seat(train,row,column,0); // set seat to available.
306     }
307     show_available(train, output); // for server-side output only
308     printf("%s\n",output);
309     return 0;
310 }
311
312 int get_train(struct clientInformation* c) {
313     // determine which train to interact with based on provided date.
314     char date[50];
315     int train;
316     GetTodayDate(date);
317     printf("customer date: %s\n",c->DateOfTravel);
318     printf("%s\n",date);
319     if (strcmp(c->DateOfTravel,date) == 0) train = 1; // train 1 is for today.
320     else {
321         GetTomorrowDate(date);
322         printf("%s\n",date);
323         if (strcmp(c->DateOfTravel,date) == 0) train = 2; // train 2 is for tomorrow.
324         else train = -1; // invalid date.
325     }
326     printf("train %d\n" train);

```

```

327     printf("train read\n",sem);
328     return train;
329 }
330 int signal_read(int train) { // separate function for signaling semaphore.
331     if (train != 0) return 0;
332     char sem_name[25];
333     strcpy(sem_name,"/summary_read"); // semaphore only for adjust readcount.
334     sem_t* sem;
335     if ((sem = sem_open(sem_name, O_RDWR)) == SEM_FAILED) {
336         printf("failed to open read semaphore for summary.\nerror number:%d",errno);
337         exit(1);
338     }
339     sem_post(sem);
340     return 0;
341 }
342
343 int wait_read(int train) { // separate function for waiting for semaphore.
344     if (train > 0) return 0;
345     char sem_name[25];
346     strcpy(sem_name,"/summary_read"); // semaphore only for adjust readcount.
347     sem_t* sem;
348     if ((sem = sem_open(sem_name, O_RDWR)) == SEM_FAILED) {
349         printf("failed to open read semaphore for summary.\nerror number:%d",errno);
350         exit(1);
351     }
352     sem_wait(sem);
353     return 0;
354 }
355
356 int signal_write(int train) { // separate function for signaling semaphore.
357     char sem_name[25];
358     if (train > 0) snprintf(sem_name,25,"/train%d",train);
359     else if (train == SUMMARY) strcpy(sem_name,"/summary_write");
360     sem_t* sem;
361     if ((sem = sem_open(sem_name, O_RDWR)) == SEM_FAILED) {
362         printf("failed to open write semaphore for train%d.\nerror numububer:%d",train,errno);
363         exit(1);
364     }
365     sem_post(sem);
366     return 0;
367 }
368
369 int wait_write(int train) { // separate function for waiting for semaphore.
370     char sem_name[25];
371     if (train > 0) snprintf(sem_name,25,"/train%d",train);
372     else if (train == SUMMARY) strcpy(sem_name,"/summary_write");
373     sem_t* sem;
374     if ((sem = sem_open(sem_name, O_RDWR)) == SEM_FAILED) {
375         printf("failed to open write semaphore for train%d.\nerror number:%d",train,errno);
376         exit(1);
377     }
378     sem_wait(sem);
379     return 0;
380 }
381
382 int check_thread_permission(int id, int train, int seats, int* seats_for_thread) {
383     seats_for_thread[id+(train-1)*NUM_THREADS] = seats; // post requested seats to the shared array.
384     int largest;
385     if (seats <= 0) return -1;
386     if (train <= 0) return -1;
387     while (1) {
388         wait_write(train); // wait for current thread to finish with train file.
389         largest = 1;
390         for (int i=0; i<NUM_THREADS; i++) { // see if this thread has largest number of requested seats.
391             if (seats_for_thread[id+(train-1)*NUM_THREADS] < seats_for_thread[i+(train-1)*NUM_THREADS]) {
392                 largest = 0;
393                 break;
394             }
395         }
396         if (largest == 1) return 0; // proceed.
397         signal_write(train); // release semaphore if thread is not chosen.
398         sleep(1); // try again in 1 second.
399     }
400 }
401
402 int serve_customer(int socket, int t_id, int s_id, int* seats_for_thread) {
403     const struct clientInformation empty_struct;
404     struct clientInformation c;
405     char m[1000];
406     int first = 1;
407     while (1) {
408         c = empty_struct; // reset customer struct when back to menu.
409         c.server = s_id; // set server id.
410         if (first) { // initial greeting.
411             snprintf(m,1000,"OHello! My name is THREAD-%d, How may I assist you today?\n\t1. Make a reservation.\n\t2. Inquiry about a ticket.\n\t3. Modify the reservation.\n\t4. Cancel the reservation.\n\t5. E:
412             first = 0;
413         } else { // back to menu message.
414             strcpy(m,"0\nIs there anything else I can help you with today?\n\t1. Make a reservation.\n\t2. Inquiry about a ticket.\n\t3. Modify the reservation.\n\t4. Cancel the reservation.\n\t5. Exit the program.\n")
415         }
416         send(socket, &m, sizeof(m), MSG_NOSIGNAL);
417         read(socket, &m, sizeof(m));
418         sscanf(m,"%d",&c.MenuOption); // scan menu option from customer.
419         printf("%d\n",c.MenuOption);
420         if (c.MenuOption == 5) {
421             strcpy(m,"2Exiting...Thank you and have a good day!\n"); // client will terminate socket.
422             send(socket, &m, sizeof(m), MSG_NOSIGNAL);
423             return 0; // thread frees up.
424         }
425         if (c.MenuOption == 1) { // make reservation
426             if (get_customer_info(socket,&c) == -1) continue; // fill clientInformation struct.
427             char date[50];
428             int train;
429             GetTodayDate(date);
430             printf("%s\n",date);
431             if (strcmp(c.DateOfTravel,date) == 0) train = 1; // train 1 = today.
432             else {
433                 GetTomorrowDate(date);
434                 if (strcmp(c.DateOfTravel,date) == 0) train = 2; // train 2 = tomorrow.
435                 else train = 1;

```

```

436     }
437 }
438 if (train == -1) {
439     strcpy(m, "1Sorry, there is no train available for the selected date.\nReservation cancelled.\n");
440     send(socket, &m, sizeof(m), MSG_NOSIGNAL);
441     return -1;
442 }
443 strcpy(m, "1Please wait...\n");
444 send(socket, &m, sizeof(m), MSG_NOSIGNAL);
445 strcpy(m, "");
446 if (check_thread_permission(t_id, train, c.NumberOfTravelers, seats_for_thread) == -1) continue; // thread priority
447 // at this point the thread has the train semaphore.
448 if (verify_enough_seats(socket, train, &c) == -1) continue; // if failed, return to menu and release semaphore.
449 if (confirm_purchase(socket, train, &c) == -1) continue; // if failed, return to menu and release semaphore.
450 send_available_seats(socket, train, &c);
451 read(socket, &m, sizeof(m)); // read seat selection.
452 if (verify_selection(socket, train, &c, m) == -1) continue; // if failed, return to menu and release semaphore.
453 add_to_train(train, &c, m); // update train file.
454 signal_write(train); // release train semaphore.
455 seats_for_thread[t_id] = 0;
456 seats_for_thread[t_id+NUM_THREADS] = 0; // set requested seats to 0 in shared array.
457 wait_write(SUMMARY);
458 addCustomer(&c, 1); // update summary file after waiting for access.
459 signal_write(SUMMARY);
460 snprintf(m, 1000, "1Reservation confirmed! Your ticket number is %d.\n", c.ticket);
461 send(socket, &m, sizeof(m), MSG_NOSIGNAL);
462 continue; // return to menu.
463 }
464 if (c.MenuOption == 2) { // inquiry
465     if (get_customer_ticket(socket, &c) == -1) continue; // ask for ticket
466     char results[500];
467     // procedure for allowing multiple readers or one writer.
468     wait_read(SUMMARY);
469     if (change_read_count(1) == 1) wait_write(SUMMARY);
470     signal_read(SUMMARY);
471     printCustomerInfo(&c, results); // read customer info from summary file. populates results string.
472     wait_read(SUMMARY);
473     if (change_read_count(-1) == 0) signal_write(SUMMARY);
474     signal_read(SUMMARY);
475     snprintf(m, 1000, "1%s\n", results); // print inquiry results.
476     send(socket, &m, sizeof(m), MSG_NOSIGNAL);
477     continue; // return to menu.
478 }
479 if (c.MenuOption == 3) { // modify.
480     if (get_customer_ticket(socket, &c) == -1) continue; // ask for ticket number.
481     // procedure for allowing multiple readers or one writer.
482     wait_read(SUMMARY);
483     if (change_read_count(1) == 1) wait_write(SUMMARY);
484     signal_read(SUMMARY);
485     createCustomer(&c); // read customer info from summary file. populates struct.
486     wait_read(SUMMARY);
487     if (change_read_count(-1) == 0) signal_write(SUMMARY);
488     signal_read(SUMMARY);
489     int train = get_train(&c); // determine which train was used.
490     char original_seats[100];
491     strcpy(original_seats, c.seats);
492     if (train == -1) {
493         snprintf(m, 1000, "1The date for this train has passed, cannot modify reservation.\n");
494         send(socket, &m, sizeof(m), MSG_NOSIGNAL);
495         continue; // return to menu.
496     }
497     int new_number = confirm_modify(socket, &c); // confirm modify. get new number of seats.
498     if (new_number == -1) continue;
499     wait_write(train); // wait for write access.
500     remove_from_train(&c, train); // remove previous reservation from train.
501     c.NumberOfTravelers = new_number;
502     send_available_seats(socket, train, &c); // send available seats and ask for input.
503     read(socket, &m, sizeof(m)); // read new selection.
504     if (verify_selection(socket, train, &c, m) == -1) continue; // if fails, return to menu and release semaphore.
505     add_to_train(train, &c, c.seats); // add updated reservation.
506     signal_write(train);
507     snprintf(m, 1000, "1Reservation modified.\n");
508     snprintf(c.modified, 200, "Reservation modified by server %d. Original seats: [%s]", s_id, original_seats); // add note
509     wait_write(SUMMARY);
510     changeOldCustomer(&c); // update summary file.
511     signal_write(SUMMARY);
512     send(socket, &m, sizeof(m), MSG_NOSIGNAL);
513     continue; // return to menu.
514 }
515 if (c.MenuOption == 4) { // delete
516     if (get_customer_ticket(socket, &c) == -1) continue; // get ticket number
517     // procedure for allowing multiple readers or one writer.
518     wait_read(SUMMARY);
519     if (change_read_count(1) == 1) wait_write(SUMMARY);
520     signal_read(SUMMARY);
521     createCustomer(&c); // read customer info from summary file. populates struct.
522     wait_read(SUMMARY);
523     if (change_read_count(-1) == 0) signal_write(SUMMARY);
524     signal_read(SUMMARY);
525     int train = get_train(&c); // determine which train was used.
526     if (train == -1) {
527         snprintf(m, 1000, "1The date for this train has passed, no need to cancel reservation.\n");
528         send(socket, &m, sizeof(m), MSG_NOSIGNAL);
529         continue; // return to menu.
530     }
531     if (confirm_cancel(socket, &c) == -1) continue; // confirm cancel.
532     wait_write(train);
533     remove_from_train(&c, train); // remove reservation from train after acquiring semaphore.
534     signal_write(train);
535     wait_write(SUMMARY);
536     deleteCustomer(&c); // remove line from summary file after acquiring semaphore.
537     signal_write(SUMMARY);
538     snprintf(m, 1000, "1Reservation cancelled.\n");
539     send(socket, &m, sizeof(m), MSG_NOSIGNAL);
540     continue; // return to menu.
541 }
542 }
543 break; // end connection if invalid option provided.
544 }
545 return 0;
546 }

```

```

545 }
546 int thread_loop(void* args) {
547     struct customer_queue* q = (struct customer_queue*) args;
548     int id;
549     for (int i=0; i<NUM_THREADS; i++) { // get thread ID
550         if (q->threads[i] == pthread_self()) {
551             id = i;
552             break;
553         }
554     }
555     while(1) {
556         q->seats_for_thread[id] = 0;
557         q->seats_for_thread[id+NUM_THREADS] = 0; // set requested seats to 0 in shared array.
558         int my_customer = -1;
559         pthread_mutex_lock(&lock); // customer queue = critical section.
560         if (q->waiting > 0) { // if customer is waiting
561             my_customer = q->sockets[q->first]; // get customer socket descriptor
562             q->sockets[q->first] = 0; // remove socket descriptor
563             q->first = q->first + 1; // move head index
564             q->waiting = q->waiting - 1; // decrease waiting number
565         }
566         pthread_mutex_unlock(&lock);
567         if (my_customer >= 0) serve_customer(my_customer, id, q->port, q->seats_for_thread); // serve customer if exists
568     }
569 }
570
571 int create_socket(int port, struct sockaddr_in* address) {
572     // standard procedure for creating server socket with specified port number
573     int server_fd;
574     if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
575         perror("Socket failed");
576         exit(1);
577     } else {
578         printf("Server socket created\n");
579     }
580     address->sin_family = AF_INET;
581     address->sin_addr.s_addr = INADDR_ANY;
582     address->sin_port = htons(8000+port);
583     if (bind(server_fd, (struct sockaddr*) address, sizeof(*address)) < 0) {
584         perror("bind failed");
585         exit(1);
586     } else {
587         printf("Server socket bound\n");
588     }
589     if (listen(server_fd, 2) < 0) {
590         perror("listen");
591         exit(1);
592     } else {
593         printf("Server socket is listening\n");
594     }
595     return server_fd;
596 }
597
598 int initialize_semaphores_threads(struct customer_queue* q, int reset_semaphores) {
599     q->first = q->waiting = 0;
600     pthread_mutex_init(&lock, NULL);
601     for (int i=0; i<NUM_THREADS; i++) { // kickoff threads
602         if (pthread_create(&(q->threads[i]), NULL, (void *)thread_loop, (void *)q) != 0) {
603             perror("Failed to create thread");
604         }
605     }
606     if (reset_semaphores == 1) { // reset semaphores if specified in command line arguments.
607         sem_unlink("/train1");
608         sem_unlink("/train2");
609         sem_unlink("/summary_read");
610         sem_unlink("/summary_write");
611     }
612     // initialize semaphores with value = 1 if not already exist
613     if ((sem_open("/train1", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR, 1)) == SEM_FAILED) {
614         printf("failed to open semaphore for train0.\nerror number:%d",errno);
615         exit(1);
616     }
617     if ((sem_open("/train2", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR, 1)) == SEM_FAILED) {
618         printf("failed to open semaphore for train1.\nerror number:%d",errno);
619         exit(1);
620     }
621     if ((sem_open("/summary_read", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR, 1)) == SEM_FAILED) {
622         printf("failed to open semaphore for train1.\nerror number:%d",errno);
623         exit(1);
624     }
625     if ((sem_open("/summary_write", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR, 1)) == SEM_FAILED) {
626         printf("failed to open semaphore for train1.\nerror number:%d",errno);
627         exit(1);
628     }
629 }
630
631 int server_loop(int server_fd, int port, struct sockaddr_in* address, struct customer_queue* q) {
632     while(1) { // wait for new connections.
633         int addrlen = sizeof(*address);
634         int new_socket;
635         if ((new_socket = accept(server_fd, (struct sockaddr*) address, (socklen_t*) &addrlen)) < 0) {
636             perror("Could not accept connection.");
637             exit(1);
638         }
639         printf("new socket accepted.\n");
640         char m[1000];
641         pthread_mutex_lock(&lock);
642         if (q->waiting < 100) {
643             q->sockets[q->first+q->waiting] = new_socket; // add new connection to queue.
644             q->waiting = q->waiting + 1;
645             snprintf(m, 1000, "Thank you for choosing Server %d. One of our threads will be with you shortly...\n", port);
646             send(new_socket, &m, sizeof(m), 0);
647         } else { // if not room in queue.
648             snprintf(m, 1000, "2Sorry, There are already %d customers waiting to be served. Please try again later.\n", 100);
649             send(new_socket, &m, sizeof(m), 0);
650         }
651         pthread_mutex_unlock(&lock);
652     }

```

```
653     return 0;
654 }
655
656 int main(int argc, char const *argv[]) {
657     int port = 0;
658     if (argc > 1) port = atoi(argv[1]); // read server id from args
659     if (port > 4 || port < 1) {
660         printf("no valid server index provided.\n");
661         exit(1);
662     }
663     int reset = 0;
664     if (argc > 2 && strcmp(argv[2], "-r") == 0) reset = 1; // read reset specifier. [if recovering from crash]
665     struct sockaddr_in address;
666     int server_fd = create_socket(port, &address);
667     struct customer_queue q;
668     q.port = port; // to give server id to threads
669     initialize_semaphores_threads(&q, reset);
670     server_loop(server_fd, port, &address, &q);
671     return 0;
672 }
```