```c
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>
#include <string.h>
#include <time.h>
#include <assert.h>
#include "cdflib.h"
#include "random.h"

extern double round(double x);

#define ARRIVAL   1
#define DEPARTURE 2
#define OBSERVE   5

#define QUEUE_SIZE 2048


// TYPE DEFINITIONS

typedef struct event {
  int type, seqno, tripdup;
  double time, size;
  struct queue *queue;
  struct customer *customer;
} Event;

typedef struct heap {
  Event **v;
  int next, size;
} Heap;

typedef struct customer {
  int id;
  double arrival, service, departure;
} Customer;

typedef struct queue {
  double lambda, mu, rho;
  int busy;
  int length, size;
  Customer **customers;
  int first, next;
} Queue;


// GLOBAL VARIABLES

Heap *h;
Queue *queue, *done;
double gtime;
int rseed = 17;
int verbose = 1;
int num_cust = 10;
int cust_id = 1;


// RANDOM UTILITY FUNCTIONS

/* mymalloc: allocate memory and check the result */

void *mymalloc (int size) {
  void *result = malloc (size);
  assert (result != NULL);
  return result;
}

/* dump_vector: print the contents of a vector in a file */
```



```c

void dump_vector (Vector *vector, char *filename) {
  FILE *fp = fopen(filename, "w");
  print_vector_fp (vector, fp);
}

double avg_vector (Vector *vector) {
  double avg = calc_avg (vector->x, vector->n);
  return avg;
}


// EVENT

Event *make_event (int type, double time, Queue *queue) {
  Event *e = (Event *) mymalloc (sizeof (Event));
  e->type = type;
  e->time = time;
  e->queue = queue;
  return e;
}

void event_print (Event *e) {
  printf ("%.2lf\t%d\n", e->time, e->type);
}


// HEAP

Heap *make_heap (int size) {
  Heap *h = (Heap *) mymalloc (sizeof (Heap));
  h->v = (Event **) mymalloc (size * sizeof (Event *));
  h->next = 1;
  h->size = size;
  return h;
}

int heap_is_empty (Heap *h) {
  return h->next == 1;
}

void heap_resize (Heap *h) {
  h->size *= 2;
  h->v = (Event **) realloc (h->v, h->size * sizeof (Event *));
}

Event *heap_cargo (Heap *h, int i) {
  if (i>0 && i < h->next) {
    return h->v[i];
  } else {
    return NULL;
  }
}

void heap_print (Heap *h) {
  int i;
  printf ("Heap contains:\n");
  for (i=1; i<h->next; i++) {
    event_print (heap_cargo (h, i));
  }
}

int heap_compare (Heap *h, int i, int j) {
  double t1, t2;
  Event *e1 = heap_cargo (h, i);
  Event *e2 = heap_cargo (h, j);
  if (e1 == NULL) {
    if (e2 == NULL) {
      return 0;
```

```
    } else {
      return -1;
    }
  }
  if (e2 == NULL) return 1;
  t1 = e1->time;
  t2 = e2->time;
  if (t1 < t2) return 1;
  if (t2 < t1) return -1;
  return 0;
}

void heap_swap (Heap *h, int i, int j) {
  Event *temp = h->v[i];
  h->v[i] = h->v[j];
  h->v[j] = temp;
}

void heap_add (Heap *h, Event *e) {
  int parent;
  int i = h->next;

  if (i == h->size) {
    heap_resize (h);
  }

  h->v[i] = e;
  h->next++;

  while (i>1) {
    parent = i/2;
    if (heap_compare (h, parent, i) > 0) break;
    heap_swap (h, i, parent);
    i = parent;
  }
}

void heap_reheapify (Heap *h, int i) {
  int left = i*2;
  int right = i*2+1;
  int winleft, winright;

  winleft = heap_compare (h, i, left);
  winright = heap_compare (h, i, right);
  if (winleft>=0 && winright>=0) return;

  if (heap_compare (h, left, right) > 0) {
    heap_swap (h, i, left);
    heap_reheapify (h, left);
  } else {
    heap_swap (h, i, right);
    heap_reheapify (h, right);
  }
}

Event *heap_peek (Heap *h) {
  if (heap_is_empty (h)) return NULL;
  return h->v[1];
}

Event *heap_remove (Heap *h) {
  Event *result;
  if (heap_is_empty (h)) return NULL;
  result = h->v[1];

  h->next--;
  h->v[1] = h->v[h->next];
  heap_reheapify (h, 1);
  return result;
```



```
}

// QUEUE

Queue *make_queue ()
{
  Queue *queue = (Queue *) mymalloc (sizeof (Queue));
  queue->lambda = 1.0;
  queue->rho = 1.0;
  queue->size = QUEUE_SIZE;
  queue->customers = (Customer **) mymalloc (QUEUE_SIZE * sizeof (Customer *));
  queue->busy = 0;
  queue->length = 0;
  queue->first = 0;
  queue->next = 0;
  return queue;
}

int queue_is_full (Queue *queue) {
  return (queue->next+1) % queue->size == queue->first;
}

int queue_is_empty (Queue *queue) {
  return queue->next == queue->first;
}

void add_to_queue(Queue *queue, Customer *customer) {
  assert (!queue_is_full(queue));
  queue->length++;
  queue->customers[queue->next] = customer;
  queue->next = (queue->next+1) % queue->size;
}

Customer *get_from_queue (Queue *queue) {
  Customer *result;
  if (queue_is_empty (queue)) {
    return NULL;
  }
  queue->length--;
  result = queue->customers[queue->first];
  queue->first = (queue->first+1) % queue->size;
  return result;
}

// CUSTOMER

Customer *make_customer ()
{
  Customer *customer = (Customer *) mymalloc (sizeof (Customer));
  customer->id = cust_id++;
  customer->arrival = -1.0;
  customer->service = -1.0;
  customer->departure = -1.0;
  return customer;
}


// EVENT HANDLERS

/* start_service: move a customer into the service area */

void start_service (Event *e, Customer *customer) {
  Queue *queue = e->queue;

  if (verbose) {
    printf ("%g\tSTARTING %d, queue = %d\n",
            gtime, customer->id, queue->length);
  }
  queue->busy = 1;
```

```c
  // reconfigure the event and add it to the heap
  e->customer = customer;
  e->type = DEPARTURE;
  e->time = gtime + customer->service;
  heap_add (h, e);
}


void handle_arrival (Event *e) {
  Queue *queue = e->queue;
  Customer *customer = e->customer;

  if (verbose) {
    printf ("%g\tARRIVAL %d, queue = %d\n",
            gtime, customer->id, queue->length);
  }

  customer->arrival = gtime;
  customer->service = rand_exp (queue->mu);

  if (queue->busy) {
    free(e);
    add_to_queue (queue, customer);
  } else {
    start_service(e, customer);
  }
}

void handle_departure (Event *e) {
  Queue *queue = e->queue;
  Customer *customer = e->customer;

  if (verbose) {
    printf ("%g\tDEPARTURE %d, queue = %d\n",
            gtime, customer->id, queue->length);
  }

  customer->departure = gtime;
  add_to_queue (done, customer);

  customer = get_from_queue (queue);
  if (customer) {
    start_service (e, customer);
  } else {
    queue->busy = 0;
  }
}

void handle_observation (Event *e) {
}


// handle_event: dispatch events to their handlers

void handle_event (Event *e) {

  gtime = e->time;

  switch (e->type) {
  case ARRIVAL:
    handle_arrival (e);
    break;
  case DEPARTURE:
    handle_departure (e);
    break;
  case OBSERVE:
    handle_observation (e);
    break;
```



```c
  }
}


// start_queue: put all the arrivals in the heap

void start_queue (Queue *queue)
{
  int i;
  Event *e;
  double x;
  double time = 0.0;
  Vector *times = make_vector(num_cust);

  for (i=0; i<num_cust; i++) {
    x = rand_exp (queue->lambda);
    add_to_vector (times, x);
    time += x;
    e = make_event (ARRIVAL, time, queue);
    e->customer = make_customer();
    heap_add (h, e);
  }
  dump_vector(times, "interarrivals");
}


/* end_queue: at the end of the simulation, compute summary
   statistics */

void end_queue (Queue *queue) {
  int i;
  Customer *cust;
  double wait, avg_wait, pred;
  Vector *service_times = make_vector(done->length);
  Vector *wait_times = make_vector(done->length);

  for (i=0; i<done->length; i++) {
    cust = done->customers[i];
    wait = cust->departure - cust->arrival;
    add_to_vector (wait_times, wait);
    add_to_vector (service_times, cust->service);
  }
  dump_vector (wait_times, "wait_times");
  dump_vector (service_times, "service_times");
  avg_wait = avg_vector (wait_times);
  pred = 1.0 / (queue->mu - queue->lambda);
  printf ("%g\t%d\t%g\t%g\n", queue->rho, num_cust, avg_wait, pred);
  exit(0);
}


int main (int argc, char *argv[]) {
  Event *e;
  int c;

  queue = make_queue ();
  done = make_queue ();

  opterr = 0;
  while ( (c = getopt (argc, argv, "n:r:x:v:")) != -1) {
    switch (c) {
    case 'n':
      num_cust = atoi (optarg);
      break;
    case 'r':
      queue->rho = atof (optarg);
      break;
    case 'x':
      rseed = atoi (optarg);
```

```
      break;
    case 'v':
      verbose = atoi (optarg);
      break;
    }
  }

  queue->mu = 0.1;
  queue->lambda = queue->rho * queue->mu;

  if (verbose) {
    printf ("%g\t%g\t%g\n", queue->lambda, queue->mu, queue->rho);
  }

  srandom(rseed);

  h = make_heap (128);
  start_queue (queue);

  while (1) {
    e = heap_remove (h);
    if (e == NULL) {
      end_queue (queue);
    }
    handle_event (e);
  }

  return 0;
}
```