CS 481/ECE 437 Programming Assignment 2: The Elevator Lab

(This programming assignment was "borrowed" (with permission) from Prof. Dorian Arnold, who borrowed it from his master's advisor, Prof. Jim Plank.)

- **This assignment is to be done individually.**
- You may discuss this programming assignment with others but you may not share code; each person must turn in their own original work.
- You are allowed to use pre-existing code (of your creation or otherwise) for container data structures only, i.e. linked lists, hash tables, search trees, etc. However, these codes must be kept in separate files, and your comments and write-up.txt file must clearly credit the source of these files. When in doubt, ask!

Concepts: (p)threads management and synchronization

Overview

In this programming assignment, you will implement a system of elevators, that will be driven by Prof. Plank's simulation framework. The simulation driver program can be found in `elevator_skeleton.c`. You are not allowed to modify these files.

The Simulator

The simulator can be invoked as follows:

```
> elevator nfloors nelevators interarrival opentime floor_to_floor duration seed
```

- `nfloors`: Number of floors in the building (int greater than one).
- `nelevators`: Number of elevators in the building (int greater than zero).
- `interarrival`: Average interarrival time in seconds of people needing elevator service (double greater than zero).
- `opentime`: Time in seconds for an elevator to move from one floor to an adjacent floor (double greater than zero).
- `floor_to_floor`: Time in seconds for an elevator to move from one floor to an adjacent floor (double greater than zero).
- `duration`: Time in seconds to simulate (double greater than zero).
- `seed`: Seed for random number generator (int).

In the simulator, each elevator and each person is its own thread. Each elevator starts the simulation on floor 1 with its door closed and can hold an infinite number of people. People are generated randomly, with random first and last names. Each last name is appended with an ascending number, so that person names are unique. One third of the people will start at floor 1 and go to another floor. Another third will start on a floor higher than floor 1 and go to floor 1. The remaining third go from one random floor to another.

To go from one floor to another, an elevator must go to the person's initial floor and open its door. The person gets on the elevator, and the elevator must close its door. The person gets off the elevator when the elevator is at the person's destination floor with its door open. The person then leaves the simulation.

Sample Output

```
prompt> elevator_part_1 5 1 5 1 1 20 0
   0.937: Sofia Riley(0) arrives at floor 01 wanting to go to floor 04.
   0.937: Elevator 01 opening its door.
   1.937: Elevator 01 door is open.
   1.937: Sofia Riley(0) gets on elevator 01 on floor 01.
   1.937: Elevator 01 closing its door.
   2.937: Elevator 01 door is closed.
   2.937: Elevator 01 moving from floor 01 to floor 04.
   5.937: Elevator 01 arrives at floor 04.
   5.937: Elevator 01 opening its door.
   6.828: Christopher Naivete(1) arrives at floor 02 wanting to go to floor 03.
   6.937: Elevator 01 door is open.
   6.937: Sofia Riley(0) gets off elevator 01 on floor 04.
   6.937: Sofia Riley(0) is done.
   6.937: Elevator 01 closing its door.
   7.937: Elevator 01 door is closed.
   7.937: Elevator 01 moving from floor 04 to floor 02.
   9.335: Devin Exploration(2) arrives at floor 02 wanting to go to floor 01.
   9.768: Claire Describe(3) arrives at floor 03 wanting to go to floor 01.
   9.937: Elevator 01 arrives at floor 02.
   9.937: Elevator 01 opening its door.
  10.937: Elevator 01 door is open.
  10.937: Christopher Naivete(1) gets on elevator 01 on floor 02.
  10.937: Elevator 01 closing its door.
  11.937: Elevator 01 door is closed.
  11.937: Elevator 01 moving from floor 02 to floor 03.
  12.937: Elevator 01 arrives at floor 03.
  12.937: Elevator 01 opening its door.
  13.937: Elevator 01 door is open.
  13.937: Christopher Naivete(1) gets off elevator 01 on floor 03.
  13.937: Christopher Naivete(1) is done.
  13.937: Elevator 01 closing its door.
  13.972: Savannah Buffet(4) arrives at floor 01 wanting to go to floor 05.
  14.937: Elevator 01 door is closed.
  14.937: Elevator 01 moving from floor 03 to floor 02.
  15.480: Jonathan Magpie(5) arrives at floor 02 wanting to go to floor 04.
  15.609: Jason Stanch(6) arrives at floor 01 wanting to go to floor 05.
  15.937: Elevator 01 arrives at floor 02.
  15.937: Elevator 01 opening its door.
  16.937: Elevator 01 door is open.
  16.937: Devin Exploration(2) gets on elevator 01 on floor 02.
  16.937: Elevator 01 closing its door.
  17.937: Elevator 01 door is closed.
```

```
17.937: Elevator 01 moving from floor 02 to floor 01.
18.937: Elevator 01 arrives at floor 01.
18.937: Elevator 01 opening its door.
19.937: Elevator 01 door is open.
19.937: Devin Exploration(2) gets off elevator 01 on floor 01.
19.937: Devin Exploration(2) is done.
19.937: Elevator 01 closing its door.
20.000: Simulation Over.         7 Started.          3 Finished.
```

At time 0.937, the elevator opens its door. This takes a second, and at time 1.937, Sofia Riley gets on the elevator. It takes one second to close the door and four seconds to get to floor four, so at time 6.937, the elevator arrives at floor 4, and at time 7.937 its door is open and Sofia Riley gets out.

Next, it closes the door and travels to floor 2 to pick up the second person, Christopher Naivete. It arrives there with the door open at time 10.937. It closes the door and takes him to his destination, arriving at time 13.937. At that point, it travels back to floor 2 to get the third person, Devin Exploration. Note that it could have picked up Claire Describe at that point, since she is on floor 3 wanting to get to floor 1. However, this is a relatively naive simulation (perhaps written by Christopher's relatives). It arrives at floor 2 for Devin at time 16.937 and spits him out at time 19.937. By the time it gets its door closed, the simulation is over.

Simulator Structs (elevator.h)

`Elevator.h` defines three structures. The first contains the parameters for the simulation, the number of people during the simulation, and a lock for updating those variables. It also contains a void * that you can define and use.

```
typedef struct {
  int nfloors;
  int nelevators;
  double interarrival_time;   /* The interarrival time of people */
  double door_time;           /* The time to open or close a door */
  double floor_to_floor_time; /* The time to go from one floor to an adjacent floor */
  int npeople_started;        /* Stats. */
  int npeople_finished;
  pthread_mutex_t *lock;
  void *v;                    /* This is what you get to define */
} Elevator_Simulation;
```

The Elevator struct has a pointer to the main `Elevator_Simulation` struct for the simulation, and each elevator has its own (void *) that you can define and use. The lock and cond are for you to use to protect the elevator's data and to block when you have to block.

```
typedef struct {
  int id;             /* Elevator id */
  int onfloor;        /* What floor the elevator is currently on */
  int door_open;      /* Whether the door is open */
  int moving;         /* Whether the elevator is moving */
  Dllist people;      /* Dllist of people on the elevator */
  pthread_mutex_t *lock;
  pthread_cond_t *cond;
```

```
  void *v;
  Elevator_Simulation *es;
} Elevator;
```

In the Person struct also contains mostly straightforward data. The Elevator is the pointer to the elevator that will take the person to his/her destination floor. It will be set when the person gets on the elevator. When that happens, ptr is set so that the person may be deleted quickly from the elevator's list of people.

```
typedef struct {
  char *fname;           /* Person's first name */
  char *lname;           /* Person's last name */
  int from;              /* Starting floor */
  int to;                /* Ending floor */
  double arrival_time;   /* When the person will arrive at "from" */
  Elevator *e;           /* The person's elevator */
  Dllist ptr;            /* Pointer to the person's entry on the elevator's dllist */
  pthread_mutex_t *lock;
  pthread_cond_t *cond;
  void *v;
  Elevator_Simulation *es;
} Person;
```

The Simulator Driver

The `main()` procedure of the simulator executes the following sequence of instructions:

1. reads the command line arguments and sets up the simulation's data
2. calls the `initialize_simulation()` so that you can initialize your (void *) as necessary
3. creates Elevator structs for all the elevators and calls `initialize_elevator()` on each in case you want to use the elevator's (void *). Note that this is done before the elevator's thread is created.
4. after creating each elevator struct, it creates a thread calling the procedure `elevator()` with the elevator as an argument. You get to write `elevator()`.
5. after creating the elevators and their threads, the main program creates a person generator thread, which sleeps and generates people at random times according to the interarrival parameter.
6. finally, the `main()` thread sleeps until the simulation is over, at which time it prints out the final line and kills the program.

The People The person generating thread is straightforward:

1. it sleeps for a random amount of time, and then generates a random person
2. it then calls `initialize_person()` in case you want to initialize the person's (void *)
3. then it creates a thread running the procedure `person()`
4. The five main activities (procedures) a person invokes, via the `person()` procedure, defined in `elevator_skeleton.c` are:

(a) `wait_for_elevator()`: blocks until an elevator is on the person's floor with its door open. At this time, the person's e field will be set to the appropriate elevator.
(b) `get_on_elevator()`: puts the person on the elevator's linked list of people
(c) `wait_to_get_off_elevator()`: blocks until the elevator has legally moved to the person's destination floor and opened the door
(d) `get_off_elevator()`takes the person off the elevator's linked list
(e) `person_done()`: allows you to perform any final activties on the person

The Elevators `Open_door()`, `close_door()`, and `move_to_floor()` are defined in `elevator_skeleton.c`. They simulate the elevators' basic activities by calling sleeping for the appropriate amount of time, printing out information, and updating the relevant parts of the elevators' data.

Part 1: The Anti-Social Elevator

Your first solution will be a simple one like the one above, where each elevator serves each person in order. Each person, when he/she calls `wait_for_elevator()` should be appended to a global list and block. Each elevator simply removes the first person from the list and services just that person, ignoring all others. It's a bad solution, but it is good practice. For this solution, you must implement the following procedures, as defined above:

- `void initialize_simulation(Elevator_Simulation *es)`: set up the global list and a condition variable for blocking elevators.
- `void wait_for_elevator(Person *p)`: append the person to the global list. Signal the condition variable for blocking elevators. Block on the person's condition variable.
- `void wait_to_get_off_elevator(Person *p)`: Unblock the elevator's condition variable and block on the person's condition variable.
- `void person_done(Person *p)`: Unblock the elevator's condition variable.
- `void *elevator(void *arg)`: Each elevator is a while loop. Check the global list and if it's empty, block on the condition variable for blocking elevators. When the elevator gets a person to service, it moves to the appropriate floor and opens its door. It puts itself into the person's e field, then signals the person and blocks until the person wakes it up. When it wakes up, it goes to the person's destination floor, opens its door, signals the person and blocks. When the person wakes it up, it closes its door and re-executes its while loop.
- Your elevator's should call `open_door()`, `close_door()` from `move_to_floor()` appropriately. All sleeping and printing should be done in `elevator_skeleton.c`. Thus, your final program that you hand in should not do any sleeping or any printing.

Unless your simulation uses very short times, you should not see the effects of non-determinism, and your solution's output should match mine.

Part 2: A Smarter Elevator

Your second part is to implement a better solution. Ok, so this solution is marginally smarter but at least it provides higher throughput. Each elevator simply moves floor by floor from floor one to the top and back again. When an elevator reaches a floor, it appropriately unloads and loads passengers:

- any passenger that has arrived at his or her destination floor must be let off;
- any person waiting for an elevator on this floor and going in the proper direction must be let on.

The synchronization here is a little trickier, because the elevator has to wait for all people to get on and off before closing its door. Additionally, multiple elevators going in the same direction should not be loading people on the same floor at the same time. Your part 2 solution should perform as well as (within a percentage point) or better than mine.

Part 3: Your Smartest Elevator

Now, you must implement an even better elevator algorithm. It will be called elevator_part_3. Your simulator will be scored by the average over all runs of the percentage of people finished. This part of the programming assignment is worth 20% of the overall grade. You must include a README file that details the strategy you implemented for part 3.

The Test Scenarios

I will test each part of this programming assignment with the following 8 scenarios found in runs.txt:

- 10 1 .1 .1 .1 12
- 10 10 .01 .1 .1 12
- 10 10 .1 .01 .01 12
- 10 10 .05 .1 .1 12
- 10 10 .02 .2 .01 12
- 10 6 .02 .02 .02 12
- 100 25 .02 .2 .01 12
- 100 4 .01 .01 .01 12

Here are the abbreviated results from the 8 runs for my parts 1 and 2, each run is the average of 10 simulations:

Run	Parameters	Part 1 Results	Part 2 Results
1	10 1 .1 .1 .1 12	9.30/122.00 : 7.63531%	82.20/120.00 : 68.5775%
2	10 10 .01 .1 .1 12	91.90/729.00 : 12.6047%	619.00/775.70 : 79.7766%
3	10 10 .1 .01 .01 12	120.60/121.60 : 99.1824%	120.50/121.40 : 99.2603%
4	10 10 .05 .1 .1 12	90.40/214.20 : 42.2242%	190.40/224.00 : 84.9572%
5	10 10 .02 .2 .01 12	126.80/453.30 : 28.0228%	349.80/487.50 : 71.7238%
6	10 6 .02 .02 .02 12	239.70/430.50 : 55.6692%	438.20/450.70 : 97.2301%
7	100 25 .02 .2 .01 12	172.30/457.60 : 37.7095%	7.80/495.50 : 1.58294%
8	100 4 .01 .01 .01 12	54.30/814.40 : 6.67671%	698.00/866.70 : 80.5229%

(Not sure what's going on w/ elevator_part_2's 7th run. Will debug at some point.)

Auxiliary Files/Programs

You can find all referenced files in the attached: `prog2_files.tgz`.

Files you need and should care about:

- `makefile`: a makefile to build the various parts of this programming assignment, including the helper programs
- `elevator.h`: contains important procedure and structure definitions (described below)
- `elevator_skeleton.c`: the driver program and procedures you do not need to implement (described below)
- `elevator_null.c`: a solution that compiles but does not run since it has empty bodies for the functions you need to implement (described below). You probably want to use this as the starting point for `elevator_part_1.c` and `elevator_part_2.c` files containing your code.
- `reorder.c`: Re-orders the simulation output events in ascending time – since threads sometime execute out of order
- `double-check.c`: Verifies that your simulation ran correctly, e.g. people get out on their destination floors with door open. You should reorder you output before attempting to validate it.
- `dllist.h` and `dllist.c`: Plank's doubly-linked list implementation. Documentation can be found at here
- `jval.h` and `jval.c`: Plank's generic data type implementation. Documentation can be found at here
- `run_and_time.sh` and `runs.txt`: shell script to run the simulated scenarios that will be used to test your programming assignment (described below).

  ```
  UNIX> sh run_and_time.sh
  usage: sh run_and_time.sh program test-num(1-8) nruns starting-seed
  ```

  where test-num is the index of one of the eight scenarios from `runs.txt`; this file is needed by `run_and_time.sh`.

Additional files you need but may not care about:

- `fields.h` and `fields.c`: Plank's library to simplify input processing. Documentation can be found at here
- `jrb.h` and `jrb.c`: Plank's red-black tree implementation. Documentation can be found at here
- `names.h`: header file used to generate random people names
- `finesleep.h` and `finesleep.c`: Plank's procedures for sub-second sleeping.

Guides & Tips

- Routines like `open_door()`, `close_door()`, `move_to_floor()`, `get_on_elevator()`, and `get_off_elevator()` call `pthread_mutex_lock()` on `es->lock` and `e->lock` for the relevant elevator struct. **If you call these routines while holding one of these locks, your program will deadlock.**

- For my part 2, I only found it necessary to modify `elevator()`, `initialize_elevator()`, and `initialize_person()`. The other functions from part 1 remained unchanged.
- I found it useful to modularize `elevator()` into three functions: `check_for_people_to_unload()`, `check_for_people_to_load()`, and `move()`
- You may find the following gdb commands useful for thread debugging:
  - `gdb --pid=<pid>`: Attach gdb to the already running process with identifier `pid`.
  - `info threads`: print information for the current threads in the process;
  - `thread thread_num`: switch texttttgdb's context to that of thread `<thread_num>`. Any requested backtraces or variable are of the thread in `gdb`'s current context.
- You may also find `valgrind`'s "thread-checker" very useful for debugging in this programming assignment.

What to turn in

YOU MUST FOLLOW THESE INSTRUCTIONS PRECISELY.

You should turn in the following:

- **all** .c and .h files needed to build your executable files, including any you've downloaded. **Do not turn in any object files (.o) or binary executable files.**
- the single `makefile` that builds the `elevator_part1`, `elevator_part2` and `elevator_part3` programs.
- A `writeup.pdf` file that describes your solution to Part 3 and any other relevant information.

When you are ready to turn in your assignment:

- Place the requisite files in a directory named lastname_prog2 where lastname is your last name;
- Move to the parent directory that contains this programming assignmet 2 directory;
- execute the command: `tar -czf <prog2_dir>.tgz <prog2_dir>`

  where <prog2_dir> is the name of the directory containing your programming assignment 2 files.
- This will create a new file <prog2_dir>.tgz containing the contents of your <prog2_dir>. You can verify the contents of this "compressed tar file" with the following command: `tar -tzf <prog2_dir>.tgz`
- Submit the file `<prog2_dir>.tgz` via UNM Learn.