Problem Set 7: Tone Generator and Multi-File Playout

Please send back via NYU Classes

- A zip archive named as
  `PS07_<your name as FirstLast>.zip`

containing the C code files that implement all aspects of all problems.

Total points: 100

This Problem Set uses the following libraries:

sndfile library to read WAV files
PortAudio library to play audio data to speaker using a callback function
ncurses library to read key presses without waiting for CR

Wav file reference: http://soundfile.sapp.org/doc/WaveFormat/
libportaudio reference: http://www.portaudio.com/
libsndfile reference: http://www.mega-nerd.com/libsndfile/

Problem 1: Tone Generator (50 points)

`tone_generator.c`
In this problem you will generate and play a tone. You are given bash script build1.sh that compiles and links this program Add to the code in the instructor-supplied file tone_generator.c and fill in code under the comment:

```
//YOUR CODE HERE
```

Part 1, Command Line Parsing

Create a program that has the following command line usage (note that [] indicates optional arguments):

```
tone_generator [-f freq_in_Hz] [-a level_in_dBFS] [-s sampling_freq_in_Hz]
```

Set the default values to:

```
double f0= 440; // frequency of tone to play out
double fs = 48000; // sampling frequency of play out
int level_dBFS = -24; // level of play out, dBFS
```

Parse Command Line

Parse the command line and overwrite the values as appropriate. If parsing fails, print usage message that is the command line shown above and exit. Command line parsing can be done a number of ways. In one, test:

```
if (argv[i][0] == '-')
```

to determine if `-f` or `-a` or `-s` and their associated arg are present on command line. If they are present, then use either a switch statement:

```
switch (argv[i][1]) {
      case ‘f’:
            …
            break;
      …
}
```

or a set of if/else if statements:

```
if (argv[i][1] == ‘f’) {
      …
}
else if () {
      …
}
```

to test if `argv[i][1]` is equal to ‘a’, ‘f’ or ‘s’. Consider using `atoi()` to convert an arg string to a integer value. This requires `#include <stdlib.h>` . Alternatively, you test for a string match to each of the flag args

```
If strncmp(arg[i], ‘-f’, 2) == 0) {
      f0 = atoi(argv[++i]);
      …
}
```

Print Parameter Values

Print the values of `f0, fs, level_dBFS`.

Part 2, Tone Generation

Define a structure:

```
typedef struct {
      float *tone;
      unsigned int num_samp;
      unsigned int next_samp;
      unsigned int count;
} soundData;
```

And declare an instance:

```
soundData data;
```

Set Tone Parameters

```
num_samp = fs;
next_samp = 0;
count = 0;
```

Note that since the tone frequency is an integer, it is guaranteed that there is an integer number of tone cycles (i.e. wavelengths) per second. In other words, when wrapping from the last sample in a tone buffer of duration 1 second (fs samples) to the first sample, it is guaranteed that there will not be any phase discontinuity.

Convert `level_dBFS` to amplitude as:

```
double level = pow(level_dBFS/20.0);
```

Note that `pow()` requires `#include <math.h>`.

Allocate Storage for Tone

Using `malloc()`, allocate to `*tone` sufficient storage for `num_samp` samples. Do this with error checking.

Initialize Tone

Fill the `tone[]` array with a sine wave signal.

```
tone[i] = level*sin(2*PI*i*f0/fs);
```

Part 3, Tone Playout

Use the PortAudio library to enable playing the tone using a callback function. See `build.sh` for how to compile and link with PortAudio

In Main Program:

After PortAudio has been initialized and enabled, the program drops into a loop:

```
while (data.count < fs * 5) {
      sleep(1);
}
```

This monitors data.count and exits after 5 seconds of tone have played. The call to sleep(1) will not prevent the callback from occurring, but does prevent this loop from spinning and consuming CPU resources.

In Callback:

Fill buffer from tone data

In the body of the callback, copy `framesPerBuffer` floats (or `BUFFER_SIZE / sizeof(SAMPLE_TYPE)`, which is the same thing) from `tone[]` to the callback buffer `out[]` (or casted_buffer[]). The first sample to copy is at index `next_samp`. If a next sample would be beyond the end of the `tone[]` array, wrap back to the start of `tone[]` by setting the index to zero. In addition, increment `count` for each sample. After the copy, set `next_samp` to the next sample needed for the next callback. For example:

```
    j = data->next_samp;
    for (i = 0; i < framesPerBuffer; i++) {
        /* check if j should wrap */
        if (j >= data->num_samp) {
            j = 0;
        }
        /* same signal in left and right channels
         * left and right channels are interleaved
    * in buffer out[]
         */
        out[2*i] = data->tone[j]; /* left */
```

```
        out[2*i+1] = data->tone[j]; /* right */
        j++; /* increment pointer to tone[] */
        data->count++; // increment sample count
    }
```

Problem 2: Playout of WAV Files (50 points)

`play_wavfiles.c`

In this problem you will load and play WAV files. You are given Bash script build2.sh that compiles and links this program Add to the code in the instructor-supplied file `play_wavfiles.c` and fill in code under the comment blocks. Your program plays a selected one of several WAV files to the laptop speaker.

Part 1, Parse command line and open all files

Create a program that has the following command line usage:

```
main ifile_list.txt
```

where `ifile_list.txt` is a plain text file that contains a list of WAV audio files that can be played

All files must have the same sampling rate and same number of channels and be no more than MAX_CHN channels.

Parse Command Line and Open Input File

Parse the command line. If parsing fails, print an error diagnostic and exit. If successful, open ifile_list.txt and read each line which is a path to one of the WAV files that can be selected and played. Print list of input files paths with a number indicating order in the file.

```
Input files:
0          file1.wav
1          file2.wav
(etc.)
```

Open WAV Files

Use the libsndfile library to open WAV audio files and read the SNDFILE header of each input file. Use error checking and error reporting in all operations. Members of the SF_INFO structure that you will want to access are:

```
sfinfo.samplerate;
sfinfo.channels;
sfinfo.format;
```

Check that all files have the same sampling rate and same number of channels. If not, print an error and exit.

Part 2, Initialize Structure and Allocate Buffers

The example code shows the definition of the structure Buf:

```
typedef struct {
     /* libsndfile data structures */
     SNDFILE *sndfile[MAX_IFILES];
     SF_INFO sfinfo[MAX_IFILES];
     unsigned int num_chan;
     int selection;
     float *buffer;
} Buf;
```

The main program declares `iBuf` of type `Buf` and a pointer `p` to `iBuf`. Initialize the value `num_chan`. The variable selection indicates to the callback which file should be played out. Initialize this to -1 which causes callback to load zeros into the D/A output buffer.

Use ncurses example code to receive key presses and take action. Possible actions based on key pressed are:

```
1 ... N:    Switch to playing input file
            corresponding to the number pressed.
Q:          Quit the program
```

If Quit, then close all files and free all allocated storage before exiting.

In the code, `iBuf.selection` functions as a one-way information link (a 1-element FIFO) between the main thread and the callback thread. This eliminates the possibility that read/write race conditions might cause undesired output.

Part 3, Play File to Audio Output

Use PortAudio library callback function to enable playing the buffers of input file to D/A.

In Callback

Fill output buffer from file data associated with selected WAV file.
In the body of the callback,

- Read the `selection` variable
- If `selection` is `-1`, fill output buffer with zeros.
- Otherwise, `selection` is the index of the sndfile data structures and hence selects a file to play. Use `sf_readf_float()` to read `framesPerBuffer` frames of float-valued audio samples from the read position of the selected file into the callback output buffer.
- If the returned count from `sf_readf_float()` is less than `framesPerBuffer`, execute `sf_seek(sndfile, 0, SF_SEEK_SET)` to rewind to the start of audio data and then do another `sf_readf_float()` to read sufficient floats to fill the remainder of the buffer.