Program 5: Musical Chairs :: Threads

CSCI 6647 / 4547 Systems Programming
Fall 2020

1 Goals

- To implement a project with several threads.
- To use signals and condition variables for communication.
- To use locks to protect shared memory and avoid both deadlock and inconsistent data.

Musical chairs. The game of musical chairs used to be popular at children's birthday parties. Suppose there are $nKids$ children at the party. To play, the mother sets out $nKids - 1$ chairs in a circle. This loop is repeated until there is only one child left – and that child gets the prize:

- Mom starts the music and the children march in a circle around the chairs.
- Without warning, Mom stops the music.
- The children all try to get a chair.
- One child is left standing, and is out of the game.
- Mom removes a chair

2 Modeling Musical Chairs

The Model. The model consists of variables that are shared by Mom and all the Kid threads. Define a struct type with these members:

- A mutex lock to control all the other parts of the model.
- An int variable, nChairs, that stores the number of chairs in use for the current round of the game.
- A pointer to the chair array, with $(nKids - 1)$ ints. A chair will contain -1 when it is empty, or a kid ID when someone is sitting in it. This array must be locked before changing its contents, but it can be read freely whether or not it is locked. All code must be written so that it does not cause an error if two threads read the same chair at the same time.
- An int variable, nMarching, for the number of kids currently marching around the chairs.
- Two condition variables that the kid-threads will use for signaling Mom.
- A Model constructor to initialize nChairs and allocate the array of chairs. The parameter to the constructor must be the number of chairs needed, which is equal to 1 less than the number of Kids. Write a matching destructor.

Define this struct in a separate header file that is included by both main (Mom) and the Kid class.

2.1 Two Global Functions

My implementation has two global functions, one necessary and the other just nice.

Necessary. When you create a thread, you must supply the name of a global function that the thread will run. The Kid functions are in the Kid class and cannot be used directly for this purpose. Therefore, we create a dummy global function that can be used to start up a thread, and will simply call the `Kid::play()` function. The actual parameter must be a Kid*. It will come into our dummy function as type `void*` because that is how C implements generic functions. The first necessity is to cast the parameter to its actual type, `Kid*`. Then it can be used to start up a Kid function. Here is the code you need:

```
void* startThread(void* kid) {
    Kid* k = (Kid*) kid;
    k->play();
    return NULL;
}
```

The call on pthread_create should be in the Kid constructor:

```
pthread_create( &tid, NULL, startThread, (void*) this );
```

Just nice. `const char* sigName(int sig)` . The body is a switch that translates the numeric signal codes to strings and returns the string. That is, SIGQUIT is translated to "SIGQUIT". I used one case for each of the three signals we expect plus a default for "Unexpected signal".

2.2 The Kid class.

The Kid class provides a convenient way to organize all the information about each Kid. You need these private data parts:

- A pointer to the shared Model.
- The Kid's ID number (0, 1, 2, . . .) and tid.
- A signal set to define the signals a Kid will listen for.
- wantSeat – the subscript of the chair that a Kid will try to capture next.
- seatNumber – the subscript of the chair a Kid has captured on this round.

Kid functions. Any access to the condition variables and anything that changes the model must be protected by your mutex lock. Remember to put lots of cout ¡¡ or printf() statements in all parts of the code.

- A constructor that accepts a Model* and an integer ID number. Initialize the signal set to listen for SIGUSR1, SIGUSR2, and SIGQUIT. Create a thread for this Kid and store its thread id in the Kid's tid field.
- Get functions for the ID number and the tid.
- A predicate (boolean function) that returns a value indicating whether the Kid is sitting or standing up.
- A mutator, standUp(), that Mom will call at the beginning of each round. This function sets the seatNumber to -1, signifying that the child is standing up (has no seat).
- doMarch(): Called from play() when the Kid receives the SIGUSR1 signal. Calculate a random seat number and save it. Increment the nMarching in the shared model and use a condition variable to signal Mom that another kid is moving. (Mom must wake up and check the situation frequently. She needs to wait until all kids are marching before turning the music off.)

- doSit(): Called from play() when the Kid receives the SIGUSR2 signal. Find a free chair. This is not so easy, because other threads are doing the same thing at the same time, and will be trying to grab the same chairs. Roughly, you need to do this:
  - Optional: To give the last kid in line a fair chance, put each kid to sleep for a random number of milliseconds.
  - Increment the nMarching counter in the shared Model.
  - Starting with the chair number that you calculated earlier, keep checking the chairs until you find a free one. Stop searching when you find a vacant chair or have gone all the way around the circle and failed to find a chair.
  - If you failed, release the mutex, signal Mom that you are dropping out, and terminate.
  - If you found a chair, store your ID number in the chair, store the chair number as your seat number, and signal Mom that you are stopping.
  - It is necessary to keep the model locked from the time you first test for an empty chair until you are ready to stop marching. Be sure to release the lock no matter what the outcome is!
- play() : the thread's main function. An infinite for loop that waits for signals and processes them appropriately. The loop is needed because you will play multiple rounds of the game. Control will leave the loop when a thread exits for one reason or another.

2.3 Mom.

The main function represents Mom. Mom will read an integer *nKids* from the command line, representing the number of children at the party. (This number must be saved until the end of the program so that Mom can join with the right number of threads.) Mom needs to do several things to set up the game:

- Instantiate the model, which will cause it to be initialized. The argument to the constructor must be the number of chairs needed, which is equal to 1 less than nKids. No locking is necessary yet because no threads exist.
- Create and initialize an array of *nKids* Kid pointers. Then execute a loop that creates N new Kids. The arguments to the Kid constructor must be a pointer to the Model and the loop counter, which will become the kid's ID number). Each Kid object will create a thread for itself.
- In a loop, call the playOneRound() function. The parameters are a pointer to the model and a pointer to the array of Kids. Loop until there is only one chair left. Debug the playOneRound() function before you try writing this loop!
- When there is only one player left, Mom announces the winner and tells that last Kid to go home.
- The main function ends with the usual pthread_join loop and appropriate comments. Remember to pthread_exit.

PlayOneRound(Model* m, Kid* players[])
This is one of Mom's functions. Debug this function before you try playing the entire game. Write a generous number of printouts in it so that you can track what is going on. (Hint: I found printf to be easier to use than cout ¡¡).

- Print a blank line and a visible ——— divider line. This will break up the output into rounds, which is very helpful.
- Initialize all the chairs in the shared model to the empty state (-1) and the number of kids marching to 0. Requires locking.
- Initialize a local variable to the number of kids who are still in the game. You will need this to control loops. It is important to read it once at the beginning of the round and use the same number, consistently, throughout. It is much more error prone (and inefficient) to keep reading this essential number and hope that it does not change!
- Mom needs to tell the kids to reinitialize their status (get up out of the chairs). Call each Kid's standUp() function, then send each a USR1 signal. To do this, Mom needs to know the tid of each kid, which she gets by calling the Kid's getTid() function.
- Then she needs to wait until every thread has been scheduled and knows it is supposed to be marching. To avoid a busy wait, we will use a loop that tests a condition variable for this communication. Each kid will use the condition variable to signal when it starts marching. The signal will wake Mom up. She will use a loop check the shared counter in the Model, and when it reaches nKids, leave the loop. All this is necessary because signals are NOT queued. If two kids send signals between one of Mom's time-slices and the next, she will only get one signal. That is enough to wake her up, but not enough to let her know how many Kids are responding.
- My program slows down the execution to a speed I can watch by having Mom wait for a second or two before turning the music off.
- To end the music, Mom will send a USR2 signal to each of the kids. Then she must wait until all the kids stop marching. Use a loop and a condition variable, as before.
- When all the kids have stopped marching, Mom must remove a child and a chair from the game. Which child? Mom can search for the one whose seatNumber is still -1. That kid will already have stopped marching, because he figured out that there were no empty chairs. His thread has already exited. Now Mom must remove him from her array of players by swapping him with the last player in the array and decrement the number of players.

The Kids. Define a function takeTurn() for the Kids (threads) to execute. When the first USR signal is received, initialize any local variables to the starting state, download the current value of $nKidsChairs$, and compute a random chair number between 0 and $nKidsChairs - 1$. Then wait for the second signal; we will pretend that the kids are marching in a circle during this waiting time.

When the second signal is received, start with the computed random chair number and test chairs, in sequence, until an empty chair is found. Then lock the chairs and try to store the thread ID in the chair you found. Unhappily, two threads can simultaneously "spot" the same chair and lock it. So after you succeed in getting past the lock, you need to test the contents of the chair *again*: it might contain someone else's ID.

If you succeed in getting the chair and storing your number in it, wait for the next starting signal. If you fail to get that chair, keep going and hope you can find a different one. In either case, make sure to UNLOCK the chairs.