CS463 - Project Two - Game of Imperative Thrones

Due Sunday, 3/29, 11:59:59pm on Blackboard.

Table of Contents

CHANGELOG

- **Thu Mar 19 17:24:38 EDT 2020**
  - updated the deadline as a COVID-19 response.
  - corrected the output to include the number of players and the word "players" in the expected output. The description did mention it but the sample outputs didn't have it.

Project Description

We're discussing some different notions of concurrency in class. The goal of this project is to get a bit of experience writing concurrent-style programs in different languages. We will implement a musical chairs simulation in an imperative language now for Project 2, and later on in Haskell for Project 4.

Each student will attempt the task in an approach to concurrency in an imperative language. I'd suggest Java's `Thread` type if you can't decide, but some already-acceptable choices are:

- `pthreads` library in `C`
- Java's `Thread` type
- other Java approaches (if you happen to be familiar with them already):
  - Java's `java.util.concurrent.Semaphore` libary
  - Java's `java.util.concurrent.atomic` libary
  - Java's `java.util.concurrent.locks` libary

You can also pitch another approach if you are familiar/curious about one. Perhaps a different language, or another library. But you must get approval beforehand if you wish to stray away from this pre-approved listing, and you must **ask more than one week before the deadline**.

The Task: Musical Chairs

`N` players want to sit on `(N-1)` chairs. An emcee (the announcer) turns the music on and off, and comments on who loses, who wins, and so on. Each round, when the music turns off, all active players try to sit in the chairs, but one will not be able to find a chair. The emcee announces that that player has lost, and the next round begins with one fewer chairs and one fewer players, until there is one winner.

Requirements

- the number of players is the first command-line argument to the whole program. When not present, your code must default to `N = 10`.
- The emcee is a separate thread, which exists for the duration of the game. (it is okay to use the main thread for the emcee).
- Each player is a separate thread. Each player has a 'name', from `P1` through `PN`. These player threads exist for the duration of the game, they are not recreated each round.
- Each chair is a *separate* resource (e.g., an object). The chairs are named `C1` through `C(N-1)`. It must be possible for multiple players to obtain different chairs at the same time.
    - If you have any sort of global lock on all chairs (meaning that only one player at a time can access them) then you are not fulfilling the requirements of this project. A common example would be that the chairs are not just in an array, but that the only way to find a chair to access is through some method call that 'controls' the array, or if the entire array is synchronized/locked while an individual is looking for a chair.
- each round, whichever player did not manage to obtain a chair is out. The remaining player threads get to play in the next round, and the highest-numbered chair is removed. This means that we always have chairs *C1* through *C(k-1)*, but not necessarily players *P1* through *Pk*. In the last round, it's always chair *C1*, but it could/should be any two of the original players.
- it's entirely up to you to decide what algorithm your players use to find chairs. They may rely upon the numbering system of the chairs and their own numbers, or randomly try chairs, or any other strategy that you can design. As long as each player is able to find each open chair eventually, it is fine. Writing up this approach is part of the required (short) document at the end.

Notes and Suggestions

- It is okay to include some extra coordination points as necessary between the emcee and players, as long as the find-a-seat phase is initiated by changing the music and the players are able to access different chairs at the same time. In a real game of musical chairs, the emcee would somehow have to inspect all the chairs and identify the person who isn't sitting, so there's a bit of a linear computation to be done here and there.
- Though not required by for your project, I added a second command-line argument for an output file name. When omitted, everything is printed to standard out, but when present, I save the contents to that file. This was occasionally helpful to further inspect outputs from various runs of the program.
- printing interleaved messages can be difficult in various languages, because multiple threads are competing for the standard output. If you are having any issues with interlaced characters from multiple messages, then you need to introduce something to coordinate the message printing. I tend to have only one resource (or thread) in charge of actual printing, and everyone else sends messages they'd like to print to that resource whenever they'd like.
    - because most of our program ends up printing things, it's entirely possible that printing may be the bottleneck of our program. Don't worry about that, but if you

come up with a way around this, be sure to mention it in your writeup.

- It's not just expected that each run will produce varied outcomes and varied orders of sitting, but keep in mind that the two separate actions of *find-chair* and *print-sat-message* also may be further apart in time than you expect. A later *sat-message* doesn't mean that's when they sat, only that that's when the message got printed! Be careful not to assume too much from the order of printed messages when debugging.
- Much more so than in single-threaded programming, if you have a ton of debug-style print statements, they will affect the timing of your program. Removing them can absolutely uncover some nasty race condition bugs. **I highly recommend regularly running your code with no debugging statements to avoid finding race conditions at the last minute**.

Required outputs

In order to facilitate grading, you need to print these messages (and *only* these messages) to standard output, in the indicated ordering. Each one is printed on a line by itself. You may introduce blank lines as desired; they will be ignored.

- the emcee indicates the game is beginning with `BEGIN N players`. This is always the first message of the whole game, with the actual positive integer value replacing `N`, e.g. `BEGIN 5 players`.
- each round, the following happens:
  - the emcee prints "round *X*", e.g. `round 1`, `round 13`, etc. The music turns on at this point (no extra message is printed).
  - after "round" has printed and all players are ready to begin, the emcee thread prints `music off` and then immediately turns off the music. Players use this event to know they may begin searching for a seat.
  - as players find and take ownership of available chairs, messages of the form "`P1 sat in C1`" must be generated and printed. Of course the correct player and chair numbers must be used.
  - the last player realizes they have lost (perhaps by the emcee telling them), and they print their last message: `P1 lost`.
- when there is only one player left, instead of printing `round X` again, the emcee announces the winner with a message, e.g.: `P1 wins!`.
- the emcee indicates the game is over with `END`. This is always the last message of the whole game.

Sample Runs

Here are three sample runs. Of course you're not expected to exactly match the order of who sits, or match who loses each round! But when someone loses in a round they need to not show up in later rounds.

```
demo$ java P2 3
BEGIN 3 players

round 1
music off
P3 sat in C2
P2 sat in C1
P1 lost

round 2
music off
P3 sat in C1
```

```
P2 lost

P3 wins!
END
```

```
demo$ java P2 3
BEGIN 3 players

round 1
music off
P3 sat in C2
P2 sat in C1
P1 lost

round 2
music off
P2 sat in C1
P3 lost

P2 wins!
END
```

```
demo$ java P2 6
BEGIN 6 players

round 1
music off
P6 sat in C2
P4 sat in C1
P1 sat in C4
P3 sat in C5
P5 sat in C3
P2 lost

round 2
music off
P3 sat in C2
P4 sat in C4
P6 sat in C3
P1 sat in C1
P5 lost

round 3
music off
P3 sat in C1
P1 sat in C2
P4 sat in C3
P6 lost

round 4
music off
P3 sat in C2
P1 sat in C1
P4 lost

round 5
music off
P3 sat in C1
P1 lost

P3 wins!
```

```
END
demo$
```

Write-up

As a last step, you will write up a brief summary of what you experienced, in a `README.txt` file (`README.pdf` is also acceptable). We anticipate around a page at most; as long as you've answered all questions adequately, there's no minimum limit. Here are the questions to address:

1. Structure Notes:
    - what is the shared resource that represents the music? And specifically, where (file/line numbers) does the emcee turn on/off the music?
    - what do you use to represent a chair? Where (file/line) are they all created?
2. What was your strategy for players to find empty chairs? How does this ensure all chairs are eventually found?
3. For our programming task, what were the challenges that you faced? Where was there competition for resources, and where was there a need for cooperation?
4. What aspects of the task were straightforward, and which ones felt difficult or laborious?
5. What kinds of bugs did you run into – deadlock? Ordering issues? How did you attempt to inspect what was going wrong with the code? (Did you use any debuggers or anything? Not required, I'm just curious how your experience went with various languages).
6. Any extra thoughts or comments?

Testing It

I've provided a consistency checker file: p2checker.py

If you correctly generate output, this code will consume the output and confirm that it is valid or not; this allows for the expected variance in who wins each time but checks for things like players staying out of the game once they lose, correct number of rounds, and so on.

It's written in python 3; you need to give it a single command line argument that is either a filename (containing all the output from a program run), or give it "stdin" indicating it'll receive all the output (likely piped to it).

```
demo$ java P2 5 > 5.txt
demo$ python3 p2checker.py 5.txt
Everything looks good!


demo$ java P2 10 | python3 p2checker.py stdin
Everything looks good!
demo$
```

It's important that you do not print out extra output when your work is turned in, as this will trash the consistency check.

Turning it in

1. Java implementations need to have a `P2.java` class with the `main` method to run your entire program.
    - for C `pthreads` implementations, you must have a `Makefile` that builds the `p2` executable to run your entire program.
    - for other implementations, you also must have a `Makefile` build your program for us, and if the language allows, it must build an executable named `p2`. If not, give adequate notes in your `README.txt` file explaining how to run your code.
2. Create a folder named with this convention:

    ```
    gmason76_p2/
    ```

    Inside of it, place the following:

    - all source code (*COMMENTED!*)
    - `README.txt`
    - `Makefile` (required for non-Java implementations)
3. zip that folder, and submit it to the correct assignment on Blackboard.

Getting Help

Threaded programing can be surprising when we first start. Please begin the project early enough that you can ask questions.

Also, think about good early milestone versions of your program that can help you ensure certain features or behaviors of your code are working before you tackle the full program.

Just a reminder, collaboration on the project is not allowed; you can discuss concurrency issues with anyone, but once you're working on musical chairs (either implementation details or thinking through how to solve it in general), you need to restrict to yourself and the instructors of the course (GTA, professor).

Grading

Points BreakDown:

category	points
Structure	50
Performance	35
Write-up	15
Total	100

Structure: 50

These points go towards the structure and approach of your code. You still need to have compiling, runnable code that is mostly complete in order to earn all these points, but it's possible to get some points here even if you have some later issues.

- +10 - emcee runs the show via stopping music
    - just means that the player threads read some variable to begin seeking chairs

    - you need to mention this in your writeup for full credit.
- +10 - each player is its own thread
    - find the loop or whatever that creates them, and ensure that this code actually performs the playing task.
- +10 - each chair is its own resource
    - e.g., *NOT* just an array of ints, or a single int of `numChairsLeft`.There should be some linear structure holding all the individual chair resources.
    - you need to mention this is your writeup for full credit.
- +10 - multiple chairs can be accessed at the same time
    - no "global lock" (one reason we don't suggest using Python…)
- + 5 - surviving players preserved for next round
    - loser thread is finished/killed, or somehow removed from gameplay while all others continue to the next round, rather than restarting an entire cast of player threads. *Creating more player threads is not allowed, even if their numbers are preserved!*
- + 5 - handles variable # of players. (not hard-coded to ten players)

Performance: 35

These points are earned by the functionality and behavior of your code.

- +15: various sizes successfully complete the game.
- +10: specific sizes repeatedly complete the game in different ways
- +10: all printouts are precisely as required
    - especially important: no extra printing in your submitted work beyond the expected messages.

Writeup: 15

- three points per question, for the first five questions. (#6 is optional).

Common Deductions

- structure:
    - poorly- or un-commented code. (up to -10)
    - forgot to mention line numbers of emcee stopping music or chairs as resources.
    - there exists some bottleneck that all chair-acquisition attempts must go through (some "global lock" that turns things into one-at-a-time requests).
- performance:
    - extra printing other than the required print statements. *clean up your code before submitting!* (up to -10)
    - mis-ordered print statements, e.g. `sat` messages not all received/printed before the next round begins, or a `lost` message that isn't the last message of the round. (up to -10)
    - program sometimes, often, or always hangs/crashes without completing the game. (much of the performance points is at stake in this situation)
- Writeup:
    - skipped parts of question or didn't answer what was asked.

More Hints

I wrote my sample solution in Java, and took notes on these kinds of situations that you may also find in your coding experience. Here are some Java-specific coding examples to some perhaps-common issues. I might expand on this list based on students' questions/discussions on piazza.

Identifying Exception-Raisers

Identifying which thread throws an exception can be annoying. One option is to name our threads initially, with the `setName()` method:

```
// in Player constructor:
setName("Player"+this.number);
```

Here's another suggestion: catch and re-raise the exception with a more specific message at the given site, preserving the previous message via `getMessage()`:

```
try {
 ...
}
catch (IndexOutOfBoundsException e){
  String info = "Player "+name+" was waiting for music to turn off.\n";
  throw new IndexOutOfBoundsException(
      String.format("%s%s\n%s",info,e.getMessage(),this.toString()));
}
```

Another way to do this is to just give the thread a name, using inherited methods (`setName(String)`, `getName()`). I'd suggest putting something like this in your constructor first:

```
setName("Player"+id);
```

And then, any printing message can just `getName()` as part of its output.

ConcurrentModificationException Issues

Any time you are writing threaded code and you work with things like `List` types, you may run into a concurrent modification exception. Things like `ArrayList` aren't thread-safe, meaning they weren't implemented with the concept of multiple threads accessing it in mind. Because it provides no locking behavior on its own, and it notices that multiple threads are involved, you may need to change things around to use something that is thread-safe. Often, it's as simple as using an `Iterator` as your looping mechanism, and relying upon its `remove()` functionality.

So, instead of:

```
for (Player p : players) { p.start(); }
```

Do something like this:

```
for (final Iterator<Player> it = players.iterator(); it.hasNext();) {
  it.next().start();
```

```
  // or, it.remove() the current item if you want that behavior.
}
```

Surprisingly Fast/Slow Threads

Threads can get surprisingly behind or ahead, by getting paused at just the wrong time. In this project, you may find players somehow playing the next round, or still playing the previous round, unless you do something to coordinate their notion of the round! How will the emcee *broadcast* this information?

- suggestion: have players print when they are about to attempt many key actions or phases, such as when they are beginning a round. Even these can be arbitrarily delayed, but if you see them far away enough, you may realize some extra coordination is required.

Message Ordering

Similar to the fast/slow threads concepts, the order of printed messages can be problematic. By definition they are separate from the action that they are reporting upon, and thus they can be arbitrarily later than you'd expect without extra coordination. You may want to pay special attention to how you will coordinate the messages. One nice approach is to have a "logger" object that receives printing requests (somehow coordinated/synchronized), and only let that logger perform the actual printing. This also gives you another *thing* that can communicate with other threads about what's happened so far. However you manage the print ordering is fine as long as it gets done. Keep in mind that the single resource - output - is being shared with multiple threads, so this may indeed be a bottleneck. Most of our program is ultimately about printing messages, so that's unavoidable to some degree. That's not a problem for successfully completing the project.

Mailboxes

One way to get a universal communication mechanism is to implement your own mailboxes. By providing a (`synchronized`) way to `get()` and `put()` messages, and letting threads access each other's mailboxes to send messages, you can get a lot of coordination settled that way. The payload of the messages will have to be some specific type. Enumerations may suffice if you just need to signal conditions, and not values, but any type (and subtypes/implementor classe?) could do.

`volatile` choices

- Java's `volatile` keyword is useful when the following circumstances are ***all*** met:
  - where updates to its value do not depend on previous values (we can "miss" a value)
  - for a variable that multiple threads will access
- one good use of it is for "flag" variables that signal a state change from on/off, true/false, etc. *How might that fit our current project?* We still need to be concerned about a thread missing a change-and-back-to-the-original-value.
- It's not necessary/useful when only one thread can/will access the variable, or for final values (no point).