ECE/CS 5510 Multiprocessor Programming Fall 2016

Homework 1

Part I: Problems [55 points]:

Learning objective: reinforcing understanding of the problem of mutual exclusion, its definition, concurrent reasoning and coordination, Amdahl's Law.

Solve the following exercises.

1. Safety/Liveness [8 points]
   For each of the following, state whether it is a safety or liveness property. Identify the bad or good thing of interest.

   1. What goes up must come down.
   2. Two processes with critical sections *c0* and *c1* may not be in their critical sections at the same time.
   3. If two or more processes are waiting to enter their critical sections, at least one succeeds.
   4. If an interrupt occurs, then a message is printed within one second.
   5. If an interrupt occurs, then a message is printed.
   6. Clients may not retain resources forever.
   7. Two things are certain: death and taxes.
   8. It is always cloudy before it rains.

2. Producer-Consumer [8 points]
   In the producer–consumer fable, we assumed that Bob can see whether the can on Alice's windowsill is up or down. Design a producer–consumer protocol using cans and strings that works even if Bob cannot see the state of Alice's can (this is how real-world interrupt bits work).

3. Prisoners Problem (1) [8 points]
   You are one of *P* recently arrested prisoners. The warden, a deranged computer scientist, makes the following announcement:

   > You may meet together today and plan a strategy, but after today you will be in isolated cells and have no communication with one another.
   >
   > I have set up a "switch room" which contains a light switch, which is either *on* or *off*. The switch is not connected to anything.
   >
   > Every now and then, I will select one prisoner at random to enter the "switch room." This prisoner may throw the switch (from *on* to *off*, or vice-versa), or may leave the switch unchanged. Nobody else will ever enter this room.
   >
   > Each prisoner will visit the switch room arbitrarily often. More precisely, for any *N*, eventually each of you will visit the switch room at least *N* times.
   >
   > At any time, any of you may declare: "we have all visited the switch room at least once." If the claim is correct, I will set you free. If the claim is incorrect, I will feed all of you to the crocodiles. Choose wisely!

   - Devise a winning strategy when you know that the initial state of the switch is *off*.

- Devise a winning strategy when you do not know whether the initial state of the switch is *on* or *off*.

Hint: not all prisoners need to do the same thing.

4. Prisoners Problem (2) [8 points]
   The same warden has a different idea. He orders the prisoners to stand in line, and places red and blue hats on each of their heads. No prisoner knows the color of his own hat, or the color of any hat behind him, but he can see the hats of the prisoners in front. The warden starts at the back of the line and asks each prisoner to guess the color of his own hat. The prisoner can answer only “red” or “blue." If he gives the wrong answer, he is fed to the crocodiles. If he answers correctly, he is freed. Each prisoner can hear the answer of the prisoners behind him, but cannot tell whether that prisoner was correct.
   The prisoners are allowed to consult and agree on a strategy beforehand (while the warden listens in) but after being lined up, they cannot communicate any other way besides their answer of “red” or “blue.”
   Devise a strategy that allows at least $P - 1$ of P prisoners to be freed.

5. Speedup/Amdahl's law [7 points]
   Running your application on two processors yields a speedup of S_2. Use Amdahl’s law to derive a formula for S_n, the speedup on n processors, in terms of n and S_2.

6. Amdahl's law: Breaking Even [8 points]
   You have a choice between buying one uniprocessor that executes eight zillion instructions per second, or a 16-core multiprocessor where each core executes one zillion instructions per second. Using Amdahl’s law, explain how you would decide which to buy for a particular application.
   Your answer must include an analytical determination of the break-even point.

7. More Amdahl's law [8 points]
   Running a program on an N-core processor takes time T_0 to complete. Not satisfied with this, the program's author manages to optimize the sequential part and make it run nine times faster. After this optimization, the program completes in one fourth of the time, $T_0/4$, when run on the same N-core processor. What fraction of the overall execution time (i.e., equivalent execution time on a uniprocessor) did the sequential part account for, before optimization? Express your answer as a function of N. Can the scenario described in this problem happen for N=6? Why or why not?

Write your answers for Part I in a PDF file named *hw1.pdf*.

Part II: Programming assignment [45 points]

Learning objective: getting started programming in Java, re-enforcing Java concepts, getting used to reading the Java documentation, setting up Eclipse IDE.

You will write a (single-threaded) console-based film manager application, called Films. You may need to use the following Java classes: Date, DateFormat, String, ArrayList, Scanner/BufferedReader/FileReader, Matcher, and others.

Films will manage an in-memory database of watched/unwatched films. It will operate upon this database by reading and executing commands from an input file named **in.txt**. The program will also produce results, which will be written to a file named **out.txt**. For the purpose of this program, it is sufficient to store the records in a simple in-memory Java collection in the order they are added (e.g., an ArrayList would be a good choice).

Films must support the following commands. <X> represents an argument passed to the command, e.g., 5, true, 18/2/1995, "John Smith". Arguments will be given without angle brackets and will be separated by whitespace; arguments that contain whitespace must be double-quoted. Arguments will not contain escaped whitespace (e.g. '\ ') or escaped quotes (e.g. \").

LOAD <filename.csv>

- Loads the contents of database from the named file, replacing the existing contents.
- The file is in a comma-separated value (CSV) format, with one entry per line.
- Each entry matches the following format exactly.
  <ReleaseDate>,<Title>,<Director>,<Watched>
- ReleaseDate is given as MM/DD/YYYY.
- Watched is either “true” or “false”.
- Title and Director may not contain commas or double quotes.
- At the end of the file, there is exactly one blank line (i.e., the text of all entries is followed by a
- new-line character \n).
- You can assume the CSV file is always given in the correct format.
- The following line should be written to the output file, where N is the number of records loaded:
  LOAD: OK <N>

STORE <filename.csv>

- Stores the contents of the database into the named file. Overwrite if the file already exists.
- The format must be exactly as described above.
- The following line should be written to the output file (not the csv file!), where N is the number
- of records stored:
  STORE: OK <N>

CLEAR

- Clears the contents of the database.
- The following line should be written to the output file:
  CLEAR: OK

ADD <Title> <Director> <ReleaseDate> <Watched>

- Add the given entry to the database. All arguments must comply with the restrictions outlined for the LOAD command.
- Duplicate entries are not allowed, i.e., adding the same title/director combo twice results in an error. (This also applies if the films were loaded from a CSV file.)
- The following line should be written to the output file:
- ADD: OK <Title> <Director>

SHOW

- Shows (by writing to the output file) all unwatched films.

- SHOW will write the following to the output file, where M is the number of unwatched films:
  SHOW: OK <M>
  <Title1>,<Director1>,<ReleaseDate1>
  <Title2>,<Director2>,<ReleaseDate2>
  …
  <TitleM>,<DirectorM>,<ReleaseDateM>
- Records will be written in the above format, with the same restrictions as for LOAD.

UPDATE <Title> <Director> <WatchedState>
UPDATE <Title> <Director> <ReleaseDate>
UPDATE <OldTitle> <OldDirector> <NewTitle> <NewDirector>

- Updates an entry. UPDATE searches for the record identified by Title Director and updates either watched state or release date based on the supplied value (the acceptable formats for the two do not overlap so there's no issue here).
- If two new arguments are supplied (NewTitle and NewDirector), those are updated instead. (Remember to check duplicate entries are not created this way. In case of duplicates, the command should result in an error.)
- Writes the following line to the output file. If Title/Director were updated, write them instead of the old ones.
  UPDATE: OK <Title> <Director>

SEARCH <Needle>

- Search for Needle as a substring in either Title or Director.
- The following lines will be written to the output file. The format for each record is the CSV format as described for LOAD. N is the number of matching records.
  SEARCH: OK <N>
  <Title1>,<Director1>,<ReleaseDate1>,<Watched1>
  <Title2>,<Director2>,<ReleaseDate2>,<Watched2>
  …
  <TitleM>,<DirectorM>,<ReleaseDateM>,<WatchedM>

Error handling

In case of errors for an individual command, Films will write the following line instead of OK. The command will not be executed.

<CMDNAME>: ERROR <ERRCODE>

For example:

ADD: ERROR DUPLICATE_ENTRY

You must handle at least the following errors (error code shown first):

1. INVALID_DATE: Invalid dates by value or format, for example: 11/31/2011, 14/23/2008, 11-31-2011, 5/5, applesauce.
2. WRONG_ARGUMENT_COUNT: Too few or too many arguments were given to a command. For example:
   ADD John 11/11/2011
   SHOW 5 10 15
   UPDATE "John Doe" 8/5/2012

3. NOT_BOOL: An argument that should be a boolean cannot be interpreted as one. For example:
   ADD Fargo William 11/1/2013 apple
4. INVALID_DATE_OR_BOOL: Specifically for UPDATE, this is for the date/watched state when the argument is neither a valid boolean nor a valid date.
5. FILE_NOT_FOUND: Self explanatory.
   LOAD missingfile.csv
6. FILM_NOT_FOUND: When the arguments to UPDATE do not match any film in memory.
7. DUPLICATE_ENTRY: When this command would result in a duplicate entry. E.g.:
   ADD John Doe 1/1/2001 true
   ADD John Doe 2/2/2002 false (error here)
   UPDATE Mike Doe John Doe (and here)
8. UNKNOWN_COMMAND: When the command is invalid. E.g.,
   PING some_website

Example execution:

in.txt	**out.txt**
CLEAR	CLEAR: OK
ADD "John Doe" "Peter Jackson" 1/5/2001 false	ADD: OK "John Doe" "Peter Jackson"
ADD Mike Blank 1/6/2002 true	ADD: OK Mike Blank
ADD Dan "Russ White" 1/7/2003 false	ADD: OK Dan "Russ White"
SHOW	SHOW: OK 2
	John Doe,Peter Jackson,1/5/2001
	Dan,Russ White,1/7/2003
UPDATE Dan "Russ White" true	UPDATE: OK Dan "Russ White"
SHOW	SHOW: OK 1
	John Doe,Peter Jackson,1/5/2001
UPDATE Dan "Russ White" Mike Blank	UPDATE: ERROR DUPLICATE_ENTRY
STORE db.csv	STORE: OK 3

Submission instructions:

Your Java files should be all grouped in a single root folder called hw1. Make sure to give your program a package name (which must start with hw1). For example, your files may be organized as follows:

hw1/	
hw1/Films.java	`package hw1;`
hw1/Database.java	`package hw1;`
hw1/util/Errors.java	`package hw1.util;`
hw1/util/CSV.java	`package hw1.util;`

Your code should be compilable and executable on a Linux Ubuntu system with Java 1.8 as follows (according to http://stackoverflow.com/a/8769536/3365726):

```
find -name "*.java" > sources.txt
javac @sources.txt
java hw1.Films
```

Note that the `java` command must be run from the folder containing your source package(s); if using the default Eclipse project layout, that would be the `src` folder, which is also where **in.txt** needs to be.

(When running your program through Eclipse, **in.txt** will need to be in the folder one level above; that is, the folder containing `src/` and `bin/`.)

Zip up the PDF from Part I, together with the source code for Part II, and save it in a file called **hw1_<myPID>.zip**. Submit the archive as Homework 1 on the class's Canvas page before the deadline.