Lab 3: Concurrency Problems

Due date: May 24, 2022, Tuesday.

Overview and Goal

In this lab, you will gain additional familiarity writing programs involving concurrency control. You will implement a couple of classical thread synchronization problems to become more comfortable with semaphores, mutex locks, monitors, and condition variables.

Setting up

In the Docker container image provided to you, the files needed for Lab 3 can be found in `/lab3`. You should get the following files:

```
makefile
DISK
System.h
System.c
Runtime.s
Switch.s
List.h
List.c
Thread.h
Thread.c
Synch.h
Synch.c
Task1.h
Task1.c
Task2.h
Task2.c
Task3.h
Task3.c
```

For your reference, the file `Synch.c` contains two versions of our solution for mutual exclusion locks in Lab 2: `Mutex` and `Mutex2` (which implements mutex locks using binary semaphores). Please use `Mutex` in Lab 3.

In this lab, you will need to modify and submit the following files:

```
Task1.h
Task1.c
Task2.h
Task2.c
```

```
Task3.h
Task3.c
```

Do not reuse any of the files from Lab 1 or Lab 2, as they are considered out of date.

Working in your Docker container image

If you prefer to use your own computer and work locally, you will need to install Docker, as shown in Lab 1. After you have successfully installed Docker on your own computer, download the Docker image we provided to you for this lab from the course website (under the section heading "Lab 3"), called `blitz-docker.tar.gz`.

Load the Docker image as you did in Lab 1 and Lab 2:

```
docker load -i blitz-docker.tar.gz
```

Finally, run the Docker image as a Docker container by using the following command:

```
docker run -it blitz
```

After the container is running, you will see a command prompt from within the Linux container. The BLITZ tools have been preinstalled for you in `/usr/local/blitz/bin`, and files needed for Lab 3 can be found in `/lab3`. The source code for building BLITZ tools can be found in `/blitz`. Within the container, the search path environment variable has already been set up for you to use BLITZ command-line tools directly.

As you may have already tried in Lab 2, there are many other commands that may be useful for you to work with Docker containers. For example, to remove all the Docker containers, you can use the command:

```
docker rm $(docker ps -a -q)
```

To remove all the Docker images (so that you can have a clean slate to start working with something else), you may use the command:

```
docker rmi $(docker images -q)
```

To copy files from the Docker container to your host computer, use the command:

```
docker cp <containerId>:/file/path/within/container /host/path/target
```

The container ID can be found in the command prompt while you are running the container.

The command-line editor `vi` has been pre-installed for you in the Docker container. To install other editor alternatives or any other packages, you can use the following command within the container:

```
apt-get install -y <package_name>
```

To re-enter the docker container after you type **`exit`** at the command line inside the container, first obtain the container ID by using the **`docker ps -a`** command:

```
$ docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
```

```
a27382779765 blitz "/bin/bash" 19 minutes ago Exited (0) 5 seconds
ago nifty_spence
```

Then use the **`docker start`** and the **`docker exec`** command to run it again:

```
$ docker start a27382779765
a27382779765
$ docker exec –it a27382779765 /bin/bash
root@a27382779765:~#
```

You will see that changes you have previously made within that container have not been lost.

Before you load the docker image provided for this lab, it is a good idea to first remove all the containers and their images related to the previous labs:

To remove all the containers:

```
docker rm $(docker ps –a –q)
```

To remove all the images:

```
docker rmi $(docker images –q)
```

To learn more about Docker containers and images, refer to the Docker documentation.

Task 1: Implement the Dining Philosopher's Solution using a Monitor

In the first task of this lab, you are required to implement a solution to the Dining Philosopher's problem in BLITZ, using a **monitor**.

Each philosopher is simulated with a thread. Each philosopher thinks for a while, and then wants to eat. Before eating, he must pick up both of his forks. After eating, he puts down his forks. Each fork is shared between two philosophers and there are 5 philosophers and 5 forks arranged in a circle.

The forks are not modelled explicitly. A fork is only picked up by a philosopher if he can pick up both forks at the same time and begin eating. To know whether a fork is available, it is sufficient to simply look at the status of the two adjacent philosophers. Another way to state the problem is to forget about the forks altogether and stipulate that a philosopher may only eat when his two neighbours are not eating. Please feel free to refer to the **Three Easy Pieces** textbook, Chapter 31.6, for a more detailed description of this problem, as well as some ideas for solutions.

A starting framework for your solution is provided in **`Task1.c`**. We have provided some code in **`Task1.c`** to set up a thread for each philosopher. The synchronization will be controlled by a **monitor**, called **`ForkMonitor`**.

The code for each thread (philosopher) is provided for you. Please read the **`PhilosphizeAndEat`** method in **`Task1.c`**, and you should not need to change this code.

The monitor to control synchronization between the threads is implemented with a class called **`ForkMonitor`**. The following class specification of **`ForkMonitor`** is provided:

```
class ForkMonitor
  superclass Object
  fields
    status: array [5] of int      -- For each philosopher: HUNGRY,
                                  -- EATING, or THINKING
  methods
    Init ()
    PickupForks (p: int)
    PutDownForks (p: int)
    PrintAllStatus ()
endClass
```

You will need to provide the code for the **Init**, **PickupForks** and **PutDownForks** methods. Please feel free to add additional fields or methods as needed. The code for **PrintAllStatus** has been provided. You should call this method whenever you change the status of any philosopher. This method will print a line of output, so you can see what is happening.

How can you proceed from this starting point? You will need a mutex lock to protect the monitor itself. There are two main methods (**PickupForks** and **PutDownForks**), which are called by the philosopher threads. As each of these methods begins, the first thing you should do is to lock the mutex of the monitor. This will ensure that only one thread at a time is executing within the monitor. Just before each of these methods returns, it must unlock the monitor (by unlocking the monitor's mutex) so that other threads can enter the monitor code.

To add proper synchronization, you will also need to use the provided implementation of *condition variables* in the **`Condition`** class, which is provided in the **`Synch`** package. Read the **`Condition`** class in **`Synch.c`**, and try to understand how condition variables are implemented.

The BLITZ emulator has a number of parameters, and one of these is how often a timer interrupt occurs. The default value — every 5000 instructions — does not produce quite as much concurrency among the philosophers as a smaller number like 2000 or 3000. Try changing this parameter to see how it affects your programs behaviour. To change the simulation parameters, type the **sim** command into the emulator. This command will give you the option to create a file called **`.blitzrc`**. After creating this file, you can edit it by hand. The next time you run the emulator, it will use this new value. Also note that too small a value — like 1000 — will cause the program to hang.

We have provided an example of a correct output in **`DesiredOutput1.txt`** (provided to you in `/share/copy/ece353s/lab3` or in the Docker image), please examine this file and try to ensure that your output is similar.

Task 2: Solving the Sleeping Barber Problem using Semaphores and Mutex Locks

Starting from the provided **`Task2.c`** and **`Task2.h`** (by creating your own KPL class from scratch), solve the following problem — called the *Sleeping Barber* problem — with semaphores and mutex locks. You will also need to create some code to print out what happens when you run the program.

The *Sleeping Barber* problem is a classical thread synchronization problem. Imagine a hypothetical barber shop with one barber, one barber chair, and a number of chairs for waiting customers. When there are no customers, the barber sits in his chair and sleeps. As soon as a customer arrives, he either wakes up the barber or, if the barber is cutting someone else's hair, sits down in one of the vacant chairs. If all of the chairs are occupied, the newly arrived customer simply leaves.

The problem arises with attempting to coordinate this activity without any race conditions or deadlocks. For example, if a solution is incorrect, the barber could end up waiting for a customer and a customer waiting for the barber, resulting in a *deadlock*. Your solution should have no race conditions or deadlocks.

To simulate the time it takes for the barber to finish the haircut, you may simply call the **`Yield()`** method.

We have provided the following functions in the **`Task2.h`** and **`Task2.c`**:

`printExample()`: This function shows an example on how the next three functions are run.

`printBarberStatus()`: This function prints out the status of the barber, which would be **start** or **end**.

`printCustomerStatus(customer: int)`: This function prints out the status of the customer, which would be **E**, **S**, **B**, **F**, or **L**.

`printChairs()`: This function prints out the status of the chairs.

We have provided an example of a correct output in **`DesiredOutput2.txt`**, which contains additional instructions on how your output should be formatted. Please examine this file and try to ensure that your output is similar. Please note that when **start** is printed, the waiting line (which is the number of empty chairs) decreases. When **S** (sit) is printed, the waiting line increases.

More background about the Sleeping Barber Problem itself has been provided in a separate file in your source code archive: `SleepingBarberProblem.pdf`.

Task 3: Solving the Gaming Parlor Problem using a Monitor

In the third task of this lab, you will attempt to solve the following problem, called the **Gaming Parlor** problem, again using a **monitor**. Groups of customers come in to a "gaming parlor" to play games. They go to the front desk to obtain one or more dice, which are used by the group while they are

playing their game, and then returned to the front desk. The front desk is in charge of lending out the dice and collecting them after each game is finished.

The gaming parlor owns only 8 dice, which are available at the front desk before the first group comes in.

The customers can play the following games. Listed after each game in parentheses is the number of dice required to play that game.

```
Backgammon (4)
Risk (5)
Monopoly (2)
Pictionary (1)
```

You should model the front desk as a monitor with the following entry methods:

```
Request(name: char, numberOfDice: int)
Return(numberOfDice: int)
```

Model each group of customers as a thread. When a group is ready to play, it must obtain the necessary number of dice. If the required number of dice is not available, then the group (i.e., the thread) must wait. You might use a condition variable to signal that "more dice have become available."

You should model the following eight different groups. Each group plays one game, as shown below, but each group plays its game 5 times. Each group must return their dice after each game and then re-acquire the dice before playing again.

A – Backgammon
B – Backgammon
C – Risk
D – Risk
E – Monopoly
F – Monopoly
G – Pictionary
H – Pictionary

Similar to the Sleeping Barber problem, to simulate the time taken by each group to play the game, simply call the **`Yield()`** method.

This problem is a generalization of the problem of resource allocation, where (1) there are a number of resources (dice) but each is identical; (2) every requesting thread needs one or more units of the resource; (3) each requesting thread knows how many units it will need before requesting any units and that info is included in the request; (4) all units are returned before any further requests are made.

Similar to Task 1, your solution must not be subject to any race conditions. In other words, regardless of the order in which the groups make their requests and return their dice, each dice must never be allocated to more than one group at a time. It should never be the case that groups are allowed to proceed when there are too few dice. Likewise, if a group has returned its dice, other groups which are waiting must be allowed to proceed once enough dice have become available. In addition, regardless of the order in which the groups make their requests, your solution should be structured such that deadlocks can never occur.

To verify that your code is working, please insert print statements to produce output like this:

```
Initializing Thread Scheduler...
Initializing Idle Process...
A requests 4
-----------------------------Number of dice now avail = 8
A proceeds with 4
-----------------------------Number of dice now avail = 4
B requests 4
-----------------------------Number of dice now avail = 4
B proceeds with 4
-----------------------------Number of dice now avail = 0
D requests 5
-----------------------------Number of dice now avail = 0
E requests 2
-----------------------------Number of dice now avail = 0
A releases and adds back 4
-----------------------------Number of dice now avail = 4
B releases and adds back 4
-----------------------------Number of dice now avail = 8
C requests 5
-----------------------------Number of dice now avail = 8
H requests 1
-----------------------------Number of dice now avail = 8
B requests 4
-----------------------------Number of dice now avail = 8
D proceeds with 5
-----------------------------Number of dice now avail = 3
   etc.
```

This output makes it fairly easy to see what the program is doing and verify that it is correct.

We have provided four functions in the **`Task3.h`** and **`Task3.c`**:
`PrintExample()`: This function shows an example on how the next three functions are run.

`Request()`: This function will let the customer request the number of dice.

`Return()`: This function will let the customer return the number of dice.

`Print()`: This function prints the name and the arguments, and the current number of dice available.

The **`Print()`** method would be called in several places:

At the beginning of the **`Request`** method:

```
self.Print (name, "requests", numNeeded)
```

At the end of the **`Request`** method:

```
self.Print (name, "proceeds with", numNeeded)
```

In the **`Return`** method:

```
self.Print (name, "releases and adds back", numReturned)
```

We have provided an example of a correct output in **`DesiredOutput3.txt`**, please examine this file and try to ensure that your output is correct.

What to Submit

Complete all three tasks in this handout (or as many tasks as you can), and then submit **`Task1.h`**, **`Task1.c`**, **`Task2.h`**, **`Task2.c`**, **`Task3.h`** and **`Task3.c.`**

Grading for this Lab

Your submitted solution will be marked using test cases. The maximum possible mark for this lab assignment is 10.