CS 4352 - Spring 2025
Project #1
Pizza Scheduling

Overview

This project will expose students to the implementation considerations within the scheduling problem space. In class, we have discussed how operating systems balance different strategies and configurations to address the matching of processing tasks to resources. We will address a hypothetical problem of a pizzeria that uses ovens, chefs and delivery drivers as resources that need to be "scheduled" to fulfill orders. Students will create a program in Java, that leverages threading, to simulate the operation of the pizza place.

Objective

During this project the students will exercise the following learning outcomes:

- Obtain in-depth understanding of process management, threads, multithreading, remote process-call and communication (1, 2)
- Construct a program module for an operating system (2, 6)
- Become proficient system programmers (2, 6)

Detailed Requirements

The students will create a Java program that will have console-based interaction for grading purposes (i.e. it will use command line arguments). The main input to the program will be a text file with a description of online orders made to a pizza place that need to be prepared, cooked and delivered to clients. Upon calling the program we will provide the appropriate flags to configure each of the simulations.

Resources

The pizzeria has 3 types of resources:

- Chefs
- Ovens
- Drivers

We will treat each of the resources in a fashion similar to a CPU in an operating system, and thus, students will be required to manage the scheduling of pizza orders matched with the resources required. The scheduling configuration for each resource can be different per execution, but once a program run starts, that configuration will remain for the duration of the simulation.

The pizzeria offers a single variety: the very controversial Hawaiian pizza! (Is it pizza or is it not?). Each pizza will require a configurable number of minutes per chef, a different number of minutes in the oven, and a final number of minutes per driver, for delivery purposes. One chef is preparing one pizza at a time, an oven can bake one pizza at a time, but a driver can deliver a full order (i.e. all pizzas in one order). When the program is started, we will provide the number of minutes for each element.

Scheduling strategies

The scheduling for each of the resources will behave differently, but in essence they map to the strategies discussed in class.

Chefs

We will assume that sometimes we will want chefs to focus a lot, so once they start preparing a pizza they won't be "preempted" from its preparation. In other words, if they start with a pizza they will not take care of another one until they finish the current one.

However, we may also want to have them multi-task. When we do that we will have chefs "round robin" around the pizzas. When in this mode, a single pizza can be "advanced" by any chef. From an analogy perspective, this would be similar to having a process run on one core first, and a different core later.

The prior requirements mean that we will have two strategies for chefs:

- --chef-strategy=FOCUSED (no preemption)
- --chef-strategy=RR (round robin)

When we do round robin we will need to provide a single quantum value for the chefs. That represents the number of minutes the chef focuses on a single pizza, before being preempted:

- --chef-quantum=<integer number>

There will be two other flags that configure how long it takes for chefs to finish preparing pizzas (--chef-time) and how many chefs there are (--available-chefs).

Ovens

They will all be non-preemptive, since removing a pizza from an oven is ill-advised for baking purposes! Once a pizza is ready for baking (when chefs are done preparing it), there will be a general first-come first-served queue that will take the next available oven. Ovens allow for prioritization, given in the orders input file. When an oven becomes available, the pizza with the highest priority should be put in first.

Two flags configure how long it takes to bake a pizza and the number of available ovens:

- --available-ovens
- --bake-time

Drivers

One driver can deliver one (and only one) order at a time. The time it takes to deliver the order comes directly from the input file. The drivers operate the same as ovens, scheduling wise: first-come first served, respecting the priorities. When all the pizzas belonging to an order are ready, and a driver becomes available, the one with the highest priority should be delivered.

The number of drivers is configured via the --available-drivers=[number] flag at program start.

Input file

When the program is started, we will provide a flag called --input-file=[a text file] where each line represents a single order, described by 4 values that are comma-separated:

- Person
- Number of pizzas
- Delivery time (in minutes)
- Priority (zero is the highest priority and it accepts other positive integers)

Thus a file with the following contents:

```
Juan,5,15,2
Maria,3,20,0
Ashley,10,30,10
```

Would mean that we have 3 orders:

- 5 pizzas, that take 15 minutes to deliver, with priority 2, placed by Juan
- 3 pizzas, that take 20 minutes to deliver, with priority 0, placed by Maria
- 10 pizzas, that take 30 minutes to deliver, with priority 10, placed by Ahsley

Simulation mechanics

The program will simulate the passing of every minute. After each minute the program must output the following to the command line (e.g. System.out print instructions):

- One output row stating the minute into the simulation that we're at
    - “==== MINUTE 5” would show the state at the end of the fifth minute
- One output row per order that shows the name of the person, the state of the order (see below), the number of pizzas done in the current state, the number of pizzas pending to finish the current state, and the amount of minutes left on the current state across all pizzas

    - “Juan,BAKING,2,3,11” means that Juan's order is done preparing, and went into backing. In the baking phase 2 have finished baking and 3 are in the process. Summing the time remaining across all 3 baking pizzas there are 11 minutes left
    - All pizzas need to finish preparing before any of them go into baking, and all pizzas need to finish baking before they go into delivery.
- One output row per resource with chefs first, followed by ovens and drivers last. Each resource will be identified by an index, starting from 0. Each resource will show the name of the person associated with the order being serviced or “None”, if currently idle
    - “Oven0,Maria” means that the first oven has one of Maria's pizza in
    - For chefs, when they are running in round robin mode, we will add a 3rd value with the quantum amount remaining
        - “Chef3,Ashley,2” means that at the end of the current minute Chef3 still has 2 minutes left of preparation given to an Ashley's pizza
        - If chefs are not round robin, then it would just state “Chef3,Ashley”

When all pizzas have been delivered, the program can stop.

Order state

As students can tell by the description, each order is the analog to a task/process that needs to be executed. They each go through the following states:

- PENDING: No resources have ever been assigned to the pizza
- PREPARING: Assigned for the first time when a chef starts working on any of the pizzas in the order. As long as one pizza is in preparation, this state remains, regardless of whether other pizzas have finished
- CHEF_WAITING: When all pizzas are waiting for a chef
- OVEN_WAITING: After all pizzas were prepared, but the oven hasn't been assigned to any of them yet
- OVEN_PREPARING: When at least one pizza is in an oven
- DRIVER_WAITING: All pizzas are out of the ovens, but there is no delivery assignment yet
- DELIVERED: After the driver finishes delivery

These states should be the ones called out as part of the output.

Program execution

As mentioned before, we will call a single main program with several flags, that allow us to meet the requirements outlined above. These are the flags students must provide:

- --input-file: text file that contains one line per order, as described above
- --available-ovens: number of chefs at the pizza place
- --available-chefs: number of chefs at the pizza place
- --available-drivers: number of drivers available

- --bake-time: number of minutes required to bake a pizza
- --chef-time: number of minutes required by chefs to prepare a pizza
- --chef-strategy: as described above
- --chef-quantum: applicable when the chef strategy is round robin

Deliverables

Students will submit a compressed file, via Blackboard, with:

- Java class files (no object code)
- Test files
  - These should be thorough enough to convince the instructor that all requirements are met
  - We will not have unit tests, but you can think of these files as almost unit testing inputs
  - The instructor will have his own test cases for grading purposes
- PDF that documents
  - Any instructions necessary to compile and run the program
  - A class diagram with all classes created and a textual description of their assignment of responsibilities. At minimum, it would be expected that each resource is represented by a class and different strategies used have a class that maps the scheduling mechanics
  - Sample output on the console for 2 different runs. It can be related to a couple of the test files described above
  - URL to a video with a runtime demo recorded by students, using an openly available platform that does not require signing in (e.g. YouTube). This video should be less than 5 minutes
  - Detailed disclosure of AI usage (see below)

Grading considerations

Each of the requirements outlined above will be an integral part of the grading. It is important to note that the instructor will have specific test cases that will require the appropriate output of the program for validation purposes. A significant portion of the grade will be based on the correct validation of the test files, but basic operation with student-provided files can earn partial credit.

The PDF document also has a portion of the grade, in correlation with the validation of the requirements.

Students are allowed to generate code with AI tools. However, if they do so, they must comply with 3 requirements:

- Add “//” at the end of each AI generated line of code (since we're using Java, an end of line comment sequence will have no runtime effect)

- At the beginning of each function or class that was AI-helped, provide a block comment with the prompt given to the generator
- A detailed explanation in the PDF document submitted, per function and/or per class, about why the generated code works and what modifications were made

Submission details

Students will submit their project via blackboard before March 14 at 11pm. Note that the project is not expected to take 3 weeks to complete, but students are given additional time to plan around their other commitments.

Extra credit (30% additional)

All prior requirements are related to a single pizza place's simulation being handled by a single main program. Students can earn an additional 30% extra grade if they are able to handle multiple restaurants, in parallel, within the same main program, by leveraging Java threading capabilities. Each restaurant must have its own thread to run. Students will need to modify how they use the --input-file flag so that it can accept a comma-separated list of files (one per restaurant).

Students will need to propose their solution in the PDF and document, using a class diagram plus textual explanation, about how they will manage multiple restaurants. The output per minute is expected to be the same but with additional sections as follows, per minute:

```
==== MINUTE <number>
==== RESTAURANT 0
[output as described before]
==== RESTAURANT 1
[output as described before]
==== RESTAURANT 2
[output as described before]

==== MINUTE <number+1>
==== RESTAURANT 0
[output as described before]
==== RESTAURANT 1
[output as described before]
==== RESTAURANT 2
[output as described before]
```

Note that it is a requirement that all threads synchronize at the end of each minute, so that the output can be generated accordingly. This synchronization mechanism is required and should be explained in the PDF **with references to the implemented code**, in order to obtain the extra credit. Solutions that accumulate the output in string variables, for instance, and hold off on the print out until the end of the simulation will not be accepted.

Also note that the per-restaurant output generated by running individual restaurant executions should be the same as their parallelized versions.