Tigerlilly Zietz
Josh Seaman
Nick Morgan

Homework 03

---

1. Write an implementation of the Dining Philosophers program, demonstrating deadlock avoidance.

   See philosophers.py

2. Write a short paragraph explaining why your program is immune to deadlock.

   Our program is immune to deadlock because it ensures that the condition never occurs where it cannot access a resource it needs to continue. Our solution is immune to deadlock because the case never occurs where a philosopher is stuck starving and waiting for another philosopher to put their fork down who never does. Our solution avoids deadlock because we use Dijkstra's resource hierarchy algorithms to determine which philosopher is eating. Our solution ensures that all but one philosopher grabs their left fork, leaving one philosopher with two forks since there will always be one remaining fork for one philosopher's right hand. We can go around the dining room table in this manner so that every philosopher gets a chance to have two forks at once. We confirm that before a philosopher eats they go through the states of *hungry* and *thinking* to confirm that they have two forks to eat with that are available to them and only them. This avoids deadlock because it ensures that a given philosopher's left and right neighbors are both not eating. This ensures that no philosopher can pick up both forks at once, unless it's their time to dine and making sure that no philosopher who sets both forks down can pick them back up again. Therefore, one philosopher eats a time and no philosopher starves.

3. Modify the `file-processes.cpp` program from Figure 8.2 on page 338 to simulate this shell command: `tr a-z A-Z < /etc/passwd`

Write the code in C, not in C++.

See file-processes.c

4. Write a program that opens a file in read-only mode and maps the entire file into the virtual-memory address space using mmap. The program should search through the bytes in the mapped region, testing whether any of them is equal to the character X. As soon as an X is found, the program should print a success message and exit. If the entire file is searched without finding an X, the program should report failure. Time your program on files of varying size, some of which have an X at the beginning, while others have an X only at the end or not at all.

See question_4.py

5. Read enough of Chapter 10 to understand the following description: In the TopicServer implementation shown in Figures 10.9 and 10.10 on pages 456 and 457, the receive method invokes each subscriber's receive method. This means the TopicServer's receive method will not return to its caller until after all of the subscribers have received the message. Consider an alternative version of the TopicServer, in which the receive method simply places the message into a temporary holding area and hence can quickly return to its caller. Meanwhile, a separate thread running in the TopicServer repeatedly loops, retrieving messages from the holding area and sending each in turn to the subscribers. What Java class from Chapter 4 would be appropriate to use for the holding area? Describe the pattern of synchronization provided by that class in terms that are specific to this particular application.

The Java class from Chapter 4 that would be appropriate to use for the holding area would be the Buffer class. The pattern of synchronization provided by the bounded buffer class enables numerous consumers and producers to share one buffer. This is relevant to this particular application because whenever the producer has a value or message ready, it can be stored in the buffer. Then, when the consumer is ready to retrieve the next value from the buffer, it will look in the immediate storage holding area, where the message is held. The buffer temporarily holds the message. The pattern of synchronization provided

by the bounded buffer class can be described in the context of this particular application as follows: The receive method places the message into a bounded buffer, queued up for later use. Meanwhile, a separate thread running in the TopicServer repeatedly loops, retrieving messages from the buffer. If there are no values available, the consumer waits. Similarly, if the buffer area is full, the producer or the receive method takes a break from storing messages in the buffer. This pattern of synchronization improves efficiency within the process.