Donovan Moini, Serena Zafiris, Ian Lizarda
Operating Systems Homework 3

1. Write an implementation of the Dining Philosophers program, demonstrating deadlock avoidance.
    a. View *dining_philosophers.py*
2. Write a short paragraph explaining why your program is immune to deadlock?
    a. The issue with the original dining philosophers problem is that whenever a philosopher has one fork and cannot pick up the other fork, the philosopher just waits until the second fork can be picked up, which eventually results in deadlock. In our program, whenever a philosopher picks up a fork and is unable to pick up the second fork, the philosopher drops the first fork, thus making it available for another philosopher to pick up, and then checks to see if both adjacent forks are available to use a few seconds later. This prevents deadlock because philosophers only use forks when they are able to use both adjacent forks, which does not keep single forks infinitely locked.
3. Modify the file-processes.cpp program from Figure 8.2 on page 338 to simulate this shell command: (*Write the code in C, not in C++.*)
   tr a-z A-Z < /etc/passwd
    a. View *tr.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.
    a. View *mmap.c*
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.

a. A bounded buffer class would be appropriate for the holding area. The way a bounded buffer works is that the producer stores values into an intermediate storage area (called the "buffer") and the consumer retrieves the value from the buffer when it is ready. In the example of the Topic Servers, the bounded buffer's pattern of synchronization can be applied like this: the receive method can store the messages into the temporary holding area, while the other thread can take the messages and deliver them to the appropriate subscribers. This way, the receive thread can be free to continue to bring messages to the holding area, while the other thread will constantly consume those messages and send them to the respective subscribers.