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

```c
#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#include <unistd.h>



//define macros

#define N 5

#define THINKING 2 //philosophers an be THINKING, HUNGRY, or EATING

#define HUNGRY 1

#define EATING 0

#define LEFT (phnum + 4) % N

#define RIGHT (phnum + 1) % N

//initializing arrays for philosophers and philosopher's state

int state[N];

int phil[N] = { 0, 1, 2, 3, 4 };



//initializing a semaphore

sem_t mutex;

//initializing an array of semaphores

sem_t S[N];



//Check: if left and right forks are available to be used

void test(int phnum)

{

   if (state[phnum] == HUNGRY

       && state[LEFT] != EATING
```

```
        && state[RIGHT] != EATING) {

        // set state of current philosopher to eating
        state[phnum] = EATING;

        //philosopher has picked up the two forks and is eating
        sleep(2);
        printf("Philosopher %d takes fork %d and %d\n",
                        phnum + 1, LEFT + 1, phnum + 1);
        printf("Philosopher %d is Eating\n", phnum + 1);
        // Sempaphore: unlock operation ->
        // sem_post(&S[phnum]) has no effect during takefork
        // used to wake up hungry philosophers during putfork
        sem_post(&S[phnum]);
    }
}
// take up forks
void take_fork(int phnum)
{
    sem_wait(&mutex);
    // change philosopher state to hungry
    state[phnum] = HUNGRY;
    printf("Philosopher %d is Hungry\n", phnum + 1);
    // eat if neighbours are not eating
    test(phnum);
    sem_post(&mutex);
    // if unable to eat wait to be signalled
    sem_wait(&S[phnum]);
```

```
    sleep(1);
}

void put_fork(int phnum)
{
    sem_wait(&mutex);
    // state that thinking
    state[phnum] = THINKING;
    printf("Philosopher %d putting fork %d and %d down\n",
           phnum + 1, LEFT + 1, phnum + 1);
    printf("Philosopher %d is thinking\n", phnum + 1);
    test(LEFT);
    test(RIGHT);
    sem_post(&mutex);
}
// philosopher class
void* philosopher(void* num)
{
    while (1) {
        int* i = num;
        sleep(1);
        take_fork(*i);
        sleep(0);
        put_fork(*i);
    }
}
```

```
//main method

int main()

{

   int i;

   pthread_t thread_id[N];

   // initialize the semaphores

   sem_init(&mutex, 0, 1);

   for (i = 0; i < N; i++)

       sem_init(&S[i], 0, 0);

   for (i = 0; i < N; i++) {

       // create philosopher processes

       pthread_create(&thread_id[i], NULL,

                      philosopher, &phil[i]);

       printf("Philosopher %d is thinking\n", i + 1);

   }

   for (i = 0; i < N; i++)



       pthread_join(thread_id[i], NULL);

}
```

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

A deadlock can occur when all philosophers decide to eat at the same time and all of them pick up their left chopstick simultaneously and/or starvation. However, in our code implementation, we represented each chopstick with a semaphore. When a philosopher tries to grab a fork/chopstick, the philosopher does so by executing a wait() operation on the semaphore, so that no other philosopher/processes can use the semaphore. The

philosopher releases teh chopsticks by executing the signal() operation on the appropriate semaphore. Therefore, we avoid deadlock by first, not allowing all philosophers to sit and eat/think at once, picking up both chopsticks, picking up both chopsticks in a critical section and alternating the choices of the first chopstick.

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 file`

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

```c
#include <unistd.h>

#include <stdio.h>

#include <fcntl.h>

#include <sys/wait.h>

#include <sys/stat.h>



//contains a program illustrating how the shell would operate.




int main() {

   pid_t returnedValue = fork(); //processid


   if(returnedValue < 0){

   printf("error forking");

   return -1;

 } else if (returnedValue == 0){

     //deallocating file descriptor

   if(close(STDOUT_FILENO) < 0){

     printf("error closing standard output");
```

```
    return -1; //exit program
  }


  if(open("etc/passwd", O_WRONLY | O_CREAT | O_TRUNC,
          S_IRUSR | S_IWUSR) < 0){
    printf("error opening my-processes");
    return -1;
  }
  execlp("tr", "tr", "a-z A-Z", NULL);
  printf("error executing tr a-z A-Z");
  return -1;


   if (waitpid(returnedValue, 0, 0) < 0) {
      printf("error waiting for child");
      return -1;
  }




}


}
```

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.

```c
#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/mman.h>

#include <string.h>


int main(int argc, char *argv[])

{

   int fd;

   char *map;

   struct stat sb;

   off_t len;

   char *p;


   if (argc < 2) {

       fprintf(stderr, "usage: %s <file>\n", argv[0]);

       exit(1);

   }


   fd = open(argv[1], O_RDONLY);

   if (fd == -1) {
```

```
    perror("open");
    exit(1);
}

if (fstat(fd, &sb) == -1) {
    perror("fstat");
    exit(1);
}

// if (fstat(fd, &amp;sb) == -1) {
//     perror("fstat");
//     exit(1);
// }

if (!S_ISREG(sb.st_mode)) {
    fprintf(stderr, "%s is not a file\n", argv[1]);
    exit(1);
}

p = mmap(0, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);
if (p == MAP_FAILED) {
    perror("mmap");
    exit(1);
}

if (close(fd) == -1) {
    perror("close");
    exit(1);
```

```
    }


    for (len = 0; len < sb.st_size; len++)
        if (p[len] == 'X') {
            printf("found at offset %lld\n", (long long)len);
            exit(0);
        }

    printf("not found\n");
    exit(1);
}
```

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.

If, instead of invoking the receive methods of the subscribers, the receive method of the TopicServer placed its message into a holding area, from where a separate thread would retrieve messages and send them to the subscribers, that holding area would best be built with a BoundedBuffer.

The pattern of synchronization used would look something like this: the sending thread repeatedly checks the buffer for new messages and waits while there is nothing to send. The main thread places messages into the buffer and immediately returns to the caller. If ever the buffer is empty, and the sending thread is waiting for a message, the main thread will notify the sending thread of the new message. Similarly, if ever the buffer reaches its arbitrary bound, and the main thread tries to feed it another message, the main thread will wait for the sending thread to catch up and send a message, opening up space in the buffer, at which point the sending thread will notify the main thread that space has been opened, and the main thread can continue.