Coby Schumitzky
Aaron Floreani
Ryan Toan Nguyen
Shaun Seah

Operating Systems Homework #3

1.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <time.h>

#define N 5 // number of philosophers
#define MAX_TIME 5 // maximum time a philosopher can spend thinking (in
seconds)

pthread_t philosophers[N];
pthread_mutex_t forks[N];
sem_t waiter;

int left(int i) { return i; } // index of philosopher to the left
int right(int i) { return (i + 1) % N; } // index of philosopher to the
right

void think(int i) {
    int thinking_time = rand() % MAX_TIME;
    printf("Philosopher %d is thinking for %d seconds\n", i,
thinking_time);
    sleep(thinking_time); // simulate thinking for random time
}

void eat(int i) {
    printf("Philosopher %d is eating\n", i);
    sleep(rand() % 3); // simulate eating for random time
}

void* philosopher(void* arg) {
    int i = *(int*)arg;
    struct timespec time_to_wait;
    while (1) {
        think(i);
        sem_wait(&waiter); // acquire the waiter semaphore
        pthread_mutex_lock(&forks[left(i)]); // pick up left fork
        pthread_mutex_lock(&forks[right(i)]); // pick up right fork
        sem_post(&waiter); // release the waiter semaphore
        eat(i);
        pthread_mutex_unlock(&forks[left(i)]); // put down left fork
        pthread_mutex_unlock(&forks[right(i)]); // put down right fork
        time_to_wait.tv_sec = rand() % MAX_TIME;
        time_to_wait.tv_nsec = 0;
        nanosleep(&time_to_wait, NULL); // sleep for random time before starting over
    }
    return NULL;
}

int main() {
    srand(time(NULL));
    int i;
    int indices[N];
    sem_init(&waiter, 0, N-1); // initialize waiter semaphore to N-1
    for (i = 0; i < N; i++) {
        pthread_mutex_init(&forks[i], NULL); // initialize fork mutexes
        indices[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher, &indices[i]); // create philosopher threads
    }
    for (i = 0; i < N; i++) {
        pthread_join(philosophers[i], NULL); // wait for philosopher threads to exit
    }
    sem_destroy(&waiter); // destroy semaphore
    for (i = 0; i < N; i++) {
        pthread_mutex_destroy(&forks[i]); // destroy mutexes
    }
    return 0;
}
```

2. This program is immune to deadlock because it uses a semaphore (waiter) to ensure that at most N-1 philosophers can be holding their forks simultaneously, where N is the number of philosophers. This ensures that at least one philosopher will always be able to pick up both their forks and eat. Additionally, the program uses a resource hierarchy, where each philosopher always picks up their forks in the same order (left, then right). This prevents the possibility of a circular waiting scenario where each philosopher is waiting for the fork held by their neighbor. Finally, the program ensures that each philosopher releases

their forks before attempting to pick up the other fork, preventing the possibility of a philosopher holding onto one fork and waiting indefinitely for the other.

3. The code can be found in the program titled “file-processes.c”. The original program in C++ was modified so that the source code is written in C. Then, functionality for the problem was added so that upon running, the forked process simulates the terminal command “tr a-z A-Z < /etc/passwd”. This command reads the information in /etc/passwd and converts all lowercase values (a-z) to uppercase values (A-Z). Then, this converted text is outputted into the console by default, as is defined by the shell command we are simulating. Upon running this program, we can see that all information from /etc/passwd is converted successfully. As this file is highly sensitive, I did add a small prompt at the beginning to ensure that the user running the program wanted to read from this file. If they answer no, the program completes and exits. Otherwise, the program will complete the shell command and exit successfully.

4.

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>

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

 if (argc < 2) {
   printf("Error! Please enter a file name as a command line argument.\n");
   exit(1);
 }

 int fd = open(argv[1], O_RDONLY);
 struct stat st;
 fstat(fd, &st);
 char *data = mmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
 int foundX = 0;

 for (size_t i = 0; i < st.st_size; i++) {
   if (data[i] == 'X') {
     printf("Success: 'X' was found!\n");
     foundX = 1;
     break;
   }
 }
 if (!foundX) {
   printf("Failure! 'X' was not found\n");
 }

 munmap(data, st.st_size);
 close(fd);
 return 0;
}
```

5. For an alternative TopicServer implementation, the pthread mutex lock described in chapter 4 could also work as a synchronization mechanism. To use a pthread mutex lock as the alternative “holding area”, you would need to create a shared message queue and protect access to it using a mutex lock. Then, when the TopicServer receives a message, it would need to acquire the lock, add the message to the queue, and then release the lock. Meanwhile, a separate thread running in the TopicServer would acquire the lock, retrieve messages from the queue, and then release the lock, therefore keeping the TopicServer in sync.