Talia, Maya, Veronica
7 April 2020
Operating Systems
Dr. Johnson

HW #3

http://bjohnson.lmu.build/cmsi387web/homework03.html

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

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define N 5
#define EAT_TIME 2

int philosophers[N] = {0,1,2,3,4};
char *philosopher_names[N] = {"Plato", "Aristotle", "Socrates", "Descartes",
"Nietzsche"};
int forks[N] = {0,1,2,3,4};

int check = 0;

pthread_t philosopher_threads[N];

static pthread_mutex_t forks_mutex[N] = {
   PTHREAD_MUTEX_INITIALIZER,
   PTHREAD_MUTEX_INITIALIZER,
   PTHREAD_MUTEX_INITIALIZER,
   PTHREAD_MUTEX_INITIALIZER,
   PTHREAD_MUTEX_INITIALIZER
};
//= {pthread_mutex_t forks_mutex0, pthread_mutex_t forks_mutex1, pthread_mutex_t
forks_mutex2, pthread_mutex_t forks_mutex3, pthread_mutex_t forks_mutex4};

int remaining = N;
```

```
// order:

//                  Plato

//         4                     0

//    Nietzsche               Aristotle

//        3                       1
//     Descartes              Socrates

//                      2

struct philosopher {
   char* name;
   int id;
   int left_fork;
   int right_fork;
   pthread_mutex_t* left_fork_mutex;
   pthread_mutex_t* right_fork_mutex;

   int has_eaten;
};

struct philosopher p[N];

void* eat(void* input) {
   int *n = (int *)input;
   int pid = *n;
   while (1) {
       if (p[pid].has_eaten == 0) {
           pthread_mutex_lock (p[pid].left_fork_mutex);
           int err = pthread_mutex_trylock (p[pid].right_fork_mutex);
           if (!err) {
               printf ("\n%s grabbed fork %d and fork %d.\n", p[pid].name,
p[pid].right_fork, p[pid].left_fork);
               printf("%s is eating.\n", p[pid].name);
               sleep (EAT_TIME);
               p[pid].has_eaten = 1;
```

```
                check ++;
                pthread_mutex_unlock(p[pid].right_fork_mutex);
            }
            pthread_mutex_unlock(p[pid].left_fork_mutex);
        }
        if (check == N) {
            break;
        }
    }
    return 0;
}

int main (void) {
    //create philosopher structs


    for (int i = 0; i < N; i++) {
        p[i].name = philosopher_names[i];
        p[i].id = philosophers[i];
        p[i].has_eaten = 0;
        p[i].left_fork = forks[i];
        if (p[i].id != 0) {
            p[i].right_fork = forks[i-1];
        } else {
            p[i].right_fork = forks[N-1];
        }
        p[i].left_fork_mutex = &forks_mutex[p[i].left_fork];
        p[i].right_fork_mutex = &forks_mutex[p[i].right_fork];

        //printf ("\n Philosopher %d File: \n Name: %s \n Right Fork: %d \n Left Fork:
%d \n", p[i].id, p[i].name, p[i].right_fork, p[i].left_fork);
    }

    //create philosopher threads


    for (int i = 0; i < N; i++) {
        pthread_create(&philosopher_threads[i], NULL, eat, (void *)&p[i].id);
    }
```

```
   //join philosopher threads

   for (int i = 0; i < N; i++) {
       pthread_join(philosopher_threads[i], NULL);
   }

   return 0;
}
```

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

By using Resource Ordering, we have been able to prevent deadlock. Given two parameters, our program deciphers which argument occurs earlier in memory (lower address), and which occurs later, at a higher address. Instead of entering a lock at the same time, causing an infinite wait, also known as deadlock, we instead lock the argument's mutexes in the order the arguments occur in memory, avoiding deadlock by ensuring one argument's action completes before completing the other.

3. Modify the file-processes.cpp program from Figure 8.2 on page 338 to simulate this shell command:

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <sys/stat.h>

// run using ./a.out < /etc/passwd
int main()
{
 pid_t returnedValue = fork();
 if (returnedValue < 0)
 {
   perror("error forking");
   return -1;
 }
 else if (returnedValue == 0)
```

```
 {
   execlp("tr", "tr", "a-z", "A-Z", NULL);
   return -1;
 }
 else
 {
   if (waitpid(returnedValue, 0, 0) < 0)
   {
     perror("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.

```
//Run using ./a.out test.txt
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
   if (argc != 2)
   {


       return -1;
   }
```

```c
int fd_in = open(argv[1], O_RDONLY);
if (fd_in < 0)
{

    perror(argv[1]);

    return -1;
}

struct stat info;
if (fstat(fd_in, &info) < 0)
{

    perror("Error stating input file");

    return -1;
}

void *addr_in = mmap(0, info.st_size, PROT_READ, MAP_SHARED, fd_in, 0);

if (addr_in == MAP_FAILED)
{
    perror("Error mapping input file");
    return -1;
}

char *asChar = (char *)addr_in;
int dataSize = info.st_size / sizeof(asChar[0]);
// int dataSize = sizeof(info.st_size) / sizeof(asChar[0]);

for (int i = 0; i < dataSize; i++)
{
    if (asChar[i] == 'X')
    {
        printf("Success! Found X. \n");
        return 0;
    }
}
printf("Failure! X was not found. \n");
```

```
    return 0;
}
```

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.

An alternative version of the TopicServer that loops through the program and places incoming messages into a holding area, instead of waiting for all of the subscribers to receive the message, would reflect the spinlock mutex we learned in Chapter 4. Both use busy waiting. The receive method loops between the holding area and the caller, just like the spinlock loops between the register and a memory location. The Java BoundedBuffer class could serve as a holding area for this situation because everytime the TopicServer receives a message, the subscriber will store the message into the intermediate buffer storage area. When the individual subscribers are ready to read the message, they will retrieve the data from the BoundedBuffer. Only when the TopicServer itself is waiting for a message, and has an empty buffer, will it’s subscribers have to wait. Now instead of having to wait until all subscribers have received the message, we can retrieve messages as they come in.