CS 453/552: Operating Systems
Midterm Examination

Time: 110 minutes *Name :*________________________ *Total Points: 150*

- *This exam has 9 questions, for a total of 165 points.*
- *Graduate students need to answer all questions. There score will be scaled down to 150 points.*
- *Undergraduate can skip one problem worth 15 points. However, they cannot skip any problem worth 20 points.*

Question	Points	Score
1	20	
2	20	
3	15	
4	20	
5	15	
6	20	
7	20	
8	20	
9	15	
Total:	165	

1. (20 points) **Where is Neo? Where is Morpheus? Somewhere in the Matrix.** A program has text, data, heap and stack segments. Consider the following C program and determine in which segment the specified variables are allocated.

```
#define BODY_BIT_SIZE 1000000
int A[BODY_BIT_SIZE];
extern void transfer();

void booth(int xyz)
{
    int i;
    static int neo[BODY_BIT_SIZE];
    int *morpheus;
    morpheus = (int *) malloc(sizeof(int)*BODY_BIT_SIZE);
    for (i=0; i<BODY_BIT_SIZE; i++)
        morpheus[i] = neo[i];
    morpheus[0] = xyz;
    transfer();
}
int main(int argc, char *argv[])
{
    char *xyz = (char *) malloc(sizeof(char*)*BODY_BIT_SIZE;
    printf("Hello?\n");
    scanf("%d", xyz)
    booth(xyz);
}
```

- `A[100]` (2 points)
- `i` (3 points)
- `morpheus` (3 points)
- `morpheus[0]` (3 points)
- `neo[10]` (3 points)
- `argc` (3 points)
- `xyz (in booth(...))` (3 points)

2. (20 points) **The Friendly Forked Nursery**. Examine the following C program fragment. How many **new** processes are created by the following code?

```
/* process A */
/* ... */
    fork();
    fork();
    fork();
    fork();
    fork();
}
/* ... */
```

Extra Credit (5 points) The Not So Friendly Forked Nursery. How many **new** processes are created by the following code?

```
/* process A */
/* ... */
for (i=0; i<n; i++) {
    fork();
}
/* ... */
```

Extra Credit (5 points). Yogi Berra said "When you come to a fork in the road, take it." Was he prescient? Explain.

prescient: Having or showing knowledge of events before they take place.

3. (15 points) **Deja Vu!** One of the useful features of a `fork()` system call is that the child process is a copy of the parent process at the time of the call. For example, we can initialize a bunch of data structures and then fork, effectively passing those data structures to all child processes. How would you simulate this with `CreateProcess()` in the MS Windows API?

4. (20 points) **Fast and safe!** Examine the code below and use PThreads to speed it up as much as possible (without having to change the code in the called functions). Assume that `compute1()`, `compute2()` and `compute3()` are independpent. But `compute1()`, `compute2()` and `compute3()` must all be finished before starting `compute4()`. And all four calls must be finished before returning from the function `do_big_calculation`.

```
void do_big_calculation()
{
    compute1();
    compute2();
    compute3();

    compute4();
}
```

5. (15 points) **Running on empty, CPUs revving...All these threads in my mind...But I cannot find...The variables that keep me flying....** Examine the following three versions of the same code. For each version, is it thread-safe? Briefly explain each answer.

```
int total = 0;

int f1(char *cmd) {
    total += strlen(cmd);
    return total;
}
```

```
int f1(char *cmd) {
    static total = 0;

    total += strlen(cmd);
    return total;
}
```

```
/* total is defined as a local variable in a single-threaded calling function */
/* but several such threads could be calling f1 directly */
void f1(int *total, char *cmd) {
    *total += strlen(cmd);
}
```

6. (20 points) **Busy Bees**. Several concurrent processes are attempting to share an I/O device. In an attempt to achieve mutual exclusion, each process is given the following structure. (*busy* is a shared Boolean variable that is initialized to **false** before the processes start running.)

```
<code unrelated to the device use>

while (busy == true); // empty loop body

if (busy == false) busy = true;
<code to access shared device>
busy = false;

<code unrelated to the device use>
```

Does the above solution guarantee mutual exclusion. If yes, then prove it. If not, then give a suitable interleaving which causes mutual exclusion to be violated.

7. (20 points) **Producers and Consumers.** In this problem we are going to examine the producers and consumers problem. We have seen a solution that used a single queue that was protected from race conditions by making access to the queue be mutually exclusive. One queue works fine if the consumers take a while to consume and producers take a while to produce. Since CPU speeds have gone up considerably, accessing the shared queue has become the main bottleneck. To improve the situation, we decide to have multiple queues. Each producer chooses a queue at random to insert data. Each consumer also chooses a queue at random to obtain data from. The code (shown later) shows an implementation using POSIX threads and semaphores. Examine it and answer the following questions:

   - Does the code satisfy mutual exclusion while accessing the queues?

   - When this code was implemented, it did not improve the performance at all! Can you explain why?

   - How would you change the code to overcome the problem discussed in the previous part?

```
Queue Q[MAXNUM];
// operations defined on the ith queue are
//  void enqueue(Q[i], item_type obj) Enter item obj into the ith queue
//  void dequeue(item_type obj, Q[i]) dequeue from ith queue and return as obj
pthread_sem_t s, empty[MAXNUM], full[MAXNUM];

void *Producer(void *arg) {
    item_type work; int id;
    for (;;) {
        work = create_work_object();
        id = random % MAXNUM; // pick a random queue number
        sem_wait(&empty[id]);
        sem_wait(&s);
            enqueue(Q[id], work);
        sem_post(&s);
        sem_post(&full[id]);
    }
}
void *Consumer(void *arg) {
    item_type work; int id;
    for (;;) {
        id = random % MAXNUM; // pick a random queue number
        sem_wait(&full[id]);
        sem_wait(&s);
            dequeue(work, Q[id]);
        sem_post(&s);
        sem_post(&empty[id]);
        perform_work(work);
    }
}
void main() {
    const int NUM_PRODUCERS = MAXNUM; //MAXNUM is defined elsewhere
    const int NUM_CONSUMERS = MAXNUM;
    pthread_t producer_thread[MAXNUM];
    pthread_t consumer_thread[MAXNUM];

    sem_init(&s, 0, 1); // initialize semaphore to 1
    for (int i=0; i<MAXNUM; i++) {
        sem_init(&empty[i], 0, MAX_QUEUE_SIZE);
        sem_init(&full[i], 0, 0);
    }
    for (int i=0; i<NUM_PRODUCERS; i++) {
        pthread_create(&producer_thread[i], NULL, &Producer, &i);
        pthread_create(&consumer_thread[i], NULL, &Consumer, &i);
    }
    for (int i=0; i<NUM_PRODUCERS; i++) {
        pthread_join(producer_thread[i], NULL);
        pthread_join(consumer_thread[i], NULL);
    }
}
```

8. (20 points) **Making code thread-safe**. Consider the following implementation of a circular queue in C. Show how to make the implementation be thread-safe using MS Windows API. The solution needs to be valid C code. Some of the relevant Windows API prototypes are shown on the last page of this exam for your convenience.

```
#ifndef __ARRAYQUEUE
#define __ARRAYQUEUE
#include <stdio.h>
#include <stdlib.h>
#define TRUE 1
#define FALSE 0

typedef struct ArrayQueue ArrayQueue;
struct ArrayQueue {
    void **A;
    int length;
    int head;
    int tail;
    int count;
};

ArrayQueue *InitArrayQueue(ArrayQueue *Q, int size)
{
    Q = (ArrayQueue *) malloc(sizeof(ArrayQueue));
    Q->A = (void **) malloc(sizeof(void *)*size);
    Q->length = size; Q->head = 0; Q->tail = 0; Q->count = 0;
    return Q;
}

int enqueue(ArrayQueue *Q, void *x)
{
    if (Q->count == Q->length){ return FALSE; //overflow }
    Q->A[Q->tail] = x;
    Q->count++;
    Q->tail=(Q->tail + 1) % Q->length;
    return TRUE;
}

void *dequeue(ArrayQueue *Q)
{
    void *x;
    if(Q->count <= 0) { return NULL; //underflow }
    x = Q->A[Q->head];
    Q->count--;
    Q->head = (Q->head + 1) % Q->length;
    return x;
}
#endif /* __ARRAYQUEUE */
```

9. (15 points) **Let's all go and find a few needles in a lot of hay!**. Consider the following Java class that does a linear search in an unordered array. Assume that this class will be called by multiple threads. How will you make the class be thread-safe?

```
import java.util.Random;

public class NeedleInALineOfHay
{
    private int A[];
    private static int counter = 0;
    private final int MAX = 1000000;
    Random generator = new Random();

    public NeedleInALineOfHay(int n) {
        A = new int[n];
        for (int i=0; i<A.length; i++)
            A[i] = generator.nextInt(MAX);
    }

    public int search(int key) {
        for (int i=0; i<A.length; i++) {
            if (A[i] == key) {
                counter++;
                return i;
            }
        }
        return -1;
    }

    public int getCounter() {return counter;}

}
```

Extra Credit (5 points) Does your solution have a good multi-threaded performance? If not, how would you get better performance?

Notes

```
HANDLE WINAPI CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  SIZE_T dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID lpParameter,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId
);

DWORD WINAPI ThreadProc(
  LPVOID lpParameter
);

HANDLE WINAPI CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
  LONG lInitialCount,
  LONG lMaximumCount,
  LPCTSTR lpName
);

HANDLE WINAPI CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,
  BOOL bInitialOwner,
  LPCTSTR lpName
);

BOOL WINAPI ReleaseMutex( HANDLE hMutex);

BOOL WINAPI ReleaseSemaphore(
  HANDLE hSemaphore,
  LONG lReleaseCount,
  LPLONG lpPreviousCount
);

DWORD WINAPI WaitForSingleObject(
  HANDLE hHandle,
  DWORD dwMilliseconds
);

DWORD WINAPI WaitForMultipleObjects(
  DWORD nCount,
  const HANDLE* lpHandles,
  BOOL bWaitAll,
  DWORD dwMilliseconds
);

CRITICAL_SECTION cs;
InitializeCriticalSection(&cs);
EnterCriticalSection(&cs);
LeaveCriticalSection(&cs);
```

```
pthread_t tid;
int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                      void *(*start_routine) (void *), void *arg);
int pthread_join(pthread_t thread, void **retval);
void pthread_exit(void *retval);


pthread_mutex_t mutex;
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutex_attr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

sem_t sem;
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t * sem);
int sem_trywait(sem_t * sem);
int sem_post(sem_t * sem);
int sem_getvalue(sem_t * sem, int * sval);
int sem_destroy(sem_t * sem);
```