CS 453/552: Operating Systems

Final Examination (Fall 2016) (Section 1)

Time: 120 minutes *Name :*____________________ *Total Points: 150*

Please read the following carefully.

- *You get five points for writing your name above!*
- *Graduate students are required to attempt the first ten problems. Their total will be scaled down to 150 points.*
- *Undergraduate students will skip one problem worth 15 points out of the first ten problems. It should be marked clearly or we will skip one 15-point problem at random. There is no extra credit for attempting all 10 problems.*
- *Problem 11 is extra credit for all students.*
- *This exam has 11 questions, for a maximum of 165 points.*

Question	Points	Score
1	20	
2	20	
3	15	
4	15	
5	20	
6	15	
7	15	
8	15	
9	15	
10	15	
11	0	
Total:	165	

1. (20 points) **Threads and Processes**.

   (a) (10 points) Which of the following statements is true regarding the different memory segments for a process and its child process (created using the `fork()` system call)?

      1. Text segment is shared. Data, heap and stack are not shared.
      2. Text and data are shared. Heap and stack are not shared.
      3. All are shared.
      4. None are shared.

   (b) (10 points) Which of the following statements is true for *multiple threads* in a process?

      1. Heap segment is shared. Text, data, and stack are not shared.
      2. Text and data are shared. Heap and stack are not shared.
      3. Text, data and heap are shared. Stack is not shared.
      4. All are shared.

2. (20 points) **Where is Neo? Where is Trinity? Somewhere in the Matrix.** 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 *trinity;
    trinity = (int *) malloc(sizeof(int)*BODY_BIT_SIZE);
    for (i=0; i<BODY_BIT_SIZE; i++)
        trinity[i] = neo[i];
    trinity[0] = xyz;
    transfer();
}
int main(int argc, char *argv[])
{
    int xyz;
    printf("Hello?\n");
    scanf("%d", xyz)
    booth(xyz);
}
```

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

3. (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);
}
```

4. (15 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 Pthreads and semaphores. Examine it and answer the following questions:

   - (5 points) Does the code satisfy mutual exclusion while accessing the queues?

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

   - (5 points) How would you change the code to overcome the problem discussed in the previous part?

```
Queue Q[MAXNUM];
// 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
sem_t s, empty[MAXNUM], full[MAXNUM];

void Producer(int i) {
    item_type work;
    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(int i) {
    item_type work;
    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);
    }
}
```

5. (20 points) **Preventing Printer Pillow-fights**.

(a) (10 points) Consider a system consisting of n threads $P_1, P_2, \ldots, P_n$. Write a monitor (in C/PThreads) that allocates three printers `printer[0..2]` among these threads. Threads get printers in a first-come-first-served fashion (one per thread). If there is more than one printer available, then each thread wants `printer[0]` since it is the fastest. If it can't get `printer[0]`, then a thread wants `printer[1]` as it is the second fastest and `printer[2]` is the least favorite choice if the first two are not available. Threads wait on a condition variable if no printer is available.

```
const int NUM=3;
pthread_cond_t  waitForPrinter;
pthread_mutex_t mutex;
int  numberOfPrintersLeft;
enum bool {false, true};
bool  printerFree[NUM]; /* true implies free, false implies busy */

// constructor to initialize the monitor
void PrintMonitorInit(void) {
    numOfPrintersLeft = NUM;
    for (int i = 0; i < NUM; i++)
        printerFree[i] = true;
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&waitForPrinter, NULL);
}

int getPrinter() {













}
```

```
void releasePrinter(int p) {











}
```

(b) (10 points) **Print Monitor in Java**. Redo the previous problem in Java.

```
public class PrintMonitor
{
    public final int NUM=3;
    private int  numberOfPrintersLeft;
    private boolean  printerFree = new boolean[NUM];

    public PrintMonitor(void) {
        numOfPrintersLeft = NUM;
        for (int i = 0; i < NUM; i++)
            printerFree[i] = true;
    }

    public int getPrinter() {











    }
```

```
    public void releasePrinter(int p) {









    }
}


public class P implements Runnable {
    int pid;
    PrintMonitor monitor;
    public P(int pid, PrintMonitor monitor) {
        this.pid = pid;
        this.monitor = monitor;
    }
    public void run() {
        int printer;
        while (true) {
            printer = monitor.getPrinter();
            printFile(printer);
            monitor.releasePrinter(printer);
            goDoSomethingElse();
        }
    }
    /* other methods */
}

pubic static void main(String [] args) {
    PrintMonitor monitor = new PrintMonitor();
    // start the n processes in parallel
    for (int i = 0; i < n; i++) {
        new Thread(new P(i, monitor)).start();
    }
}
```

6. (15 points) **Alice in Pointerland**. The following code shows how to dynamically allocate a two-dimensional $n \times n$ array in C.

```
int **array;
array = (int **) malloc(n *sizeof(int *));
for (i=0; i<n; i++)
    array[i] = (int *) malloc(n * sizeof(int));
```

The following code also allocates a two-dimensional $n \times n$ array in C.

```
int **array;
int *ptr;

ptr = (int *) malloc(sizeof(int) * n * n);
array = (int **) malloc(sizeof(int *) * n);
for (i=0; i<n; i++)
    array[i] = ptr + i * n;
```

Draw the memory layout of the two versions of dynamically allocating a two-dimensional array ? Why would the second version be useful?

7. (15 points) **Buddy Races**. Explain how you would make the buddy system be thread safe. Provide specific details about how to handle the `buddyInit, malloc, realloc, calloc` and `free` methods. Your solution should not modify your existing code for these methods beyond changing the signature. You are allowed to add new methods.

8. (15 points) **Thrash, frash, mash**. Suppose you were observing a system that is *thrashing*, that is, having a lot of page faults. Which of the following *best* describes the state of the system?

   (a) Very high CPU usage and very high I/O activity.

   (b) Low CPU usage and very high I/O activity.

   (c) Moderate CPU usage and high I/O activity.

   (d) Very high CPU usage and low I/O activity.

9. (15 points) **Please. This should be easy!** Suppose we have a computer system with a 48-bit virtual address, 36-bit physical address and a page size of 64K.

   (a) (3 points) What is the size of a page frame?

   (b) (3 points) How many page frames are there in the physical address?

   (c) (3 points) How many pages are there in the virtual address?

   (d) (6 points) Show how a 48-bit virtual address gets mapped into a 36-bit physical address by drawing out a virtual address and specifying which bits are used to index the page table and which bits are used to determine the offset into a page. Then show the mapping to the physical address.

10. (15 points) **The blocks we build when we first practice to go big.** The following fragment of code is from `ext4.h` in the Linux kernel source code.

```
#define EXT4_MIN_BLOCK_SIZE     1024
#define EXT4_MAX_BLOCK_SIZE     65536
/*
 * Constants relative to the data blocks
 */
#define EXT4_NDIR_BLOCKS        12
#define EXT4_IND_BLOCK          EXT4_NDIR_BLOCKS
#define EXT4_DIND_BLOCK         (EXT4_IND_BLOCK + 1)
#define EXT4_TIND_BLOCK         (EXT4_DIND_BLOCK + 1)
#define EXT4_N_BLOCKS           (EXT4_TIND_BLOCK + 1)
/*
 * Structure of an inode on the disk
 */
struct ext4_inode {
    ...
    __le32   i_block[EXT4_N_BLOCKS];/* Pointers to blocks */
    ...
}
```

Assume a block size of **65536 bytes** and the addresses are **64-bit**, answer the following questions for the ext4 file system. Please express your answers in appropriate units like KB, MB, GB, TB, PB etc.

- What is the maximum size of a file using only direct pointers?

- What is the maximum size of a file using direct and single indirect pointers?

- What is the maximum size of a file using direct, single, and double indirect pointers?

- What is the maximum size of a file using direct, single, double, and triple indirect pointers?

11. (15 points) **Extra Credit: Thread Pools**. Discuss how you would implement a pool of threads in C. The idea is to have a queue of jobs that have to be executed and to have a pool of fixed number of threads that execute those jobs. What will the queue contain? How is the thread pool created and how does it execute the jobs? The user should be able to limit the number of threads when they create the thread-pool. Write out the API.

Notes

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

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_treelike(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

pthread_cond_t cond;
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```