Name (Last, First): _____

This exam consists of 6 questions on 9 pages; be sure you have the entire exam before starting. The point value of each question is indicated at its beginning; the entire exam is worth 100 points. Individual parts of a multi-part question are generally assigned approximately the same point value: exceptions are noted. This exam is open text and notes. However, you may NOT share material with another student during the exam.

Be concise and clearly indicate your answer. Presentation and simplicity of your answers may affect your grade. Answer each question in the space following the question. If you find it necessary to continue an answer elsewhere, clearly indicate the location of its continuation and label its continuation with the question number and subpart if appropriate.

You should read through all the questions first, then pace yourself.

The questions begin on the next page.

Problem	Possible	Score
1	20	
2	10	
3	20	
4	20	
5	10	
6	20	
Total	100	

1. (_____/20 points)

   Short answer questions (continued on next page)

   (a) Recall that you needed to setup the user stack for Project 2. Assume you have the following arguments as a string: "echo os is cool". What is the minimum amount of space needed to setup the stack with these arguments? Another way to ask the question is: after you setup the stack with these arguments, what is the value of `PHYS_BASE - esp`? Assume your target is a 32-bit machine.

   (b) What is the difference between a function call and a system call?

(c) Consider the following C code using UNIX system calls. How many processes, including the main process are created?

```
void main(void)
{
    int x = 1, id1, id2;

    id1 = fork();
    id2 = fork();

    if (id2 == 0) {
        x = 99;
        sleep(10);
        exit(0);
    }

    if (x == 99) {
        id = fork();
    }
    sleep(10)
}
```

(d) What are the advantages of VTRR scheduling? What are the disadvantages?

2. (_____/10 points)

Parallel Processing with fork()

Recall from Project 0 you needed to write a parallel program using fork that finds the min and max values of an array. Here a portion of the code for an incorrect solution to this problem:

```
int main(int argc, char **argv)
{
    int *array;
    int nprocs = 0, arraysize = 0, arraychunksize = 0;
    char randomstate[8];
    int id, i;

    /* process command line arguments */
    if (argc != 3) {
        printf("usage: ./findminmax <nprocs> <arraysize>\n");
        return 1;
    }

    nprocs = atoi(argv[1]);
    arraysize = atoi(argv[2]);
    arraychunksize = arraysize / nprocs;

    /* allocate array and populate with random values */
    array = (int *) malloc(sizeof(int) * arraysize);

    initstate(getpid(), randomstate, 8);
    for (i = 0; i < arraysize; i++) {
        array[i] = random();
    }

    /* spawn worker processes */
    mtf_measure_begin();
    for (i = 0; i < nprocs; i++) {
        id = fork();
        if (id == 0) {
            find_min_and_max(array + (i * arraychunksize), arraychunksize);
            exit(0);
        }
        waitpid(id, NULL, 0);
    }
    mtf_measure_end();
    mtf_measure_print_seconds(1);
    return 0;
}
```

What is wrong with this solution? Show how to fix the problem. Provide changes and new code. Also, give an English explanation of your solution. You can assume that `find_min_and_max(int *a, int n)` works properly.

3. (_____/20 points)

User-level Memory Allocation

Recall your implementation of the user-level memory allocator API from Project 0. Consider the following C code that uses your API:

```
/* include files omitted */
int main(int argc, char *argv[])
{
        uint8_t *a, *b, *c, *d;

        printf("sizeof(struct used_block) = %u\n", sizeof(struct used_block));
        printf("sizeof(struct free_block) = %u\n", sizeof(struct free_block));
        mem_init(memory, 128);
        a = mem_alloc(64);
        printf("a_rel = %u\n", (unsigned) (a - memory));
        b = mem_alloc(16);
        printf("b_rel = %u\n", (unsigned) (b - memory));
        c = mem_alloc(16);
        printf("c_rel = %u\n", (unsigned) (c - memory));
        d = mem_alloc(16);
        printf("d_rel = %u\n", (unsigned) (d - memory));
        mem_free(c);
        mem_free(a);
        a = mem_alloc(32);
        printf("a_rel = %u\n", (unsigned) (a - memory));
        exit(0);
}
```

Here is the output from running this program on a 32-bit machine:

```
sizeof(struct used_block) = 4
sizeof(struct free_block) = 12
a_rel = 64
b_rel = 44
c_rel = *X*
d_rel = 4
a_rel = *Y*
```

Determine the values of `*X*` and `*Y*`. Assume you allocator splits used blocks from the top of free blocks and that you have implemented a first-fit allocation policy. Show your work.

4. (_____/20 points)

Recursive Locks

Sometimes it is useful to have *recursive* locks. A recursive lock is one that can be acquired several times by the lock holder. It can also be released by the lock holder, but it is not really released until the outermost release is called. For example, if a thread calls `lock_acquire(&a)` 3 times, the lock will not be released until `lock_release(&a)` is called 3 times. Currently Pintos locks are not recursive. Modify the data structure and function below so that locks can be initialized and used as recursive locks. If a lock is not initialize as recursive, then it is considered a normal lock.

```
struct lock {
    struct thread *holder;      /* Thread holding lock (for debugging). */
    struct semaphore semaphore; /* Binary semaphore controlling access. */


};


void lock_init (struct lock *lock, bool recursive)
{
  ASSERT (lock != NULL);

  lock->holder = NULL;
  sema_init (&lock->semaphore, 1);


}

void lock_acquire (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (!lock_held_by_current_thread (lock));


  sema_down (&lock->semaphore);
  lock->holder = thread_current ();


}

void lock_release (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (lock_held_by_current_thread (lock));


  lock->holder = NULL;
  sema_up (&lock->semaphore);


}
```

5. (_____/10 points)

Priority Inversion and Priority Donation

Assume you have properly implemented priority scheduling and priority donation in the Pintos kernel. Determine the output of the following test code (assume that the main thread is running at `PRI_DEFAULT`):

```
static void
a_thread_func (void *lock_)
{
  struct lock *lock = lock_;

  msg ("AAA");
  lock_acquire (lock);
  msg ("AAA Lock a");
  lock_release (lock);
}

static void
b_thread_func (void *lock_)
{
  struct lock *lock = lock_;

  msg ("BBB");
  lock_acquire (lock);
  msg ("BBB Lock b");
  lock_release (lock);
}

void
main (void)
{
  struct lock a, b;

  lock_init (&a);
  lock_init (&b);

  lock_acquire (&a);
  lock_acquire (&b);

  msg ("Main 1");
  thread_create ("a", PRI_DEFAULT + 1, a_thread_func, &a);
  msg ("Main 2");

  thread_create ("b", PRI_DEFAULT + 2, b_thread_func, &b);
  msg ("Main 3");

  lock_release (&b);
  msg ("Main 4");
  lock_release (&a);
  msg ("Main 5");
}
```

6. (_____/20 points)

Implementing Pipes in Pintos

Currently, the system calls you implemented for Project 2 do not support UNIX style pipes. For this problem you need to design an implementation for unix pipes on top of your Project 2 solution. You do not need to provide source code, but you need to provide enough detail and pseudo code such that an working implementation could be built using your design. Please consider the following issues: any new system calls needed, modifications to the file descriptor table, the design of the pipe buffers, the synchronization needed to support the pipe.

Note that Pintos does not support the `fork()` system call so you need to devise some way to allow a parent and child to share the pipe file descriptors.

Continue your answers here if necessary.