1> RPC Concepts & Definitions

What is RPC? Privilege separation?

What is stub code? What is marshalling?

What is server stub code? What is unmarshalling?

2> Implementing RPC in C. Simple stub code example:

3> How do you marshal an int? float? struct? Linked list? Graph?

4> What is IDL (Interface Design Language)?

5> Complexity and latency of RPC vs local calls?

6> Working with structured data

Transferring large amounts of structured data:
JSON vs xml vs Google Protocol Buffers

7> Challenge: What argument(s) to this program will cause it to print "Admin/Debug rights"?

```
#define N (20)
int admin, debug;
int histogram[N];

static int hash(char* str) {
   int c, h = 0; // sdbm hash
   while (c = *str++)
       h = c + (h << 6) + (h << 16) - h;
   return h;
}

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

   while(argc>1) {
      char*word= argv[ --argc];
      int h = hash(word);
      histogram[ (h<0?-h:h) % N ] ++;
   }
   if(admin || debug) puts("Admin/Debug rights");
   return;
}
```

> Midterm 2 "Revenge of malloc MP"

i) Two parallel programs. The parent process will report the on exit condition of the two child processes e.g. exit normally or exited due to a signal.

ii) Implement a fixed-size memory allocator. Block coalescing and block splitting will not be required.

You will complete the functions `allocate`, `release` and associated placement function in the file `memorypool.c`. These functions will behave like `malloc` and `free` respectively except that allocate memory from an external, fixed size memory pool. Your code will not call `malloc`, `sbrk` etc, instead, when `allocate` requires more memory it will call our function `void* growpool(size_t extra)` which behaves like `sbrk` except it returns `NULL` if no more memory is available. The pool is contiguous (you can assume that it is a simple `char` array).

A very simple (but incomplete!) allocator would just request more memory for each `allocate` request:

```
void* allocate(size_t size){
  void* mem = growpool(size); // NULL if we are less than
size bytes from the end of the pool
  return mem;
}
void release(void*ptr) { /* do nothing */}
```

```
typedef struct Block {
 struct Block *next;
 size_t size;  // # of usable bytes in this block
 int is_free; // 0 (allocated) or 1 (available)
} Block;
```

The linked list will contain all currently-allocated blocks and available free blocks. A simple O(N) search of all of these blocks will have sufficient performance.

The graded tests assume that the block entry is immediately before the available memory region. For example,

```
int main(){
  void* mem = allocate(100);
  // internally this will call growpool( 100+sizeof(Block) )
  // The internal linked list now includes this allocation

  Block* b = mem - sizeof(Block);
// Expect b->size to be 100,

// b->is_free to be 0
// (and b points to the first byte of the memory pool)

 release(mem);
// Expect b->size to be 100, b->is_free to be 1

  void* ptr = allocate(64);
// allocate can reuse the existing block!
// Expect b->size to be 100, b->is_free to be 0 and ptr == mem
}
```

A makefile, test cases and starting code will be provided.

Grading

Your grade will be based on the uploaded versions of `memorypool.c` and `par.c`. These files must compile and run using unmodified versions of the othe files.Use the makefile and review the output. You can scroll back on the terminal or open the file `score.txt`**.** The source code for the grading unit tests is not provided. If the testing code crashes you may assume the bug is in YOUR code. Their output may give you hints about which part of the specification is incorrectly implemented. You are expected to be able to find errors in your own code by careful analysis of your C code. You are free to use `gdb` and the memory.c to create simple tests but reading code is usually more productive.

Submit your work Each time you run `make` **without an argument** it will update the file `midterm2_upload.zip` to include a copy of your source files and your score report. Before logging off be sure to complete the following *three* steps: Upload `midterm2_upload.zip`to PrairieLearn; Submit the PraireLearn question; Finish the quiz.