Remarks: Keep the answers compact, yet precise and to-the-point. Long-winded answers that do not address the key points are of limited value. Binary answers that give little indication of understanding are no good either. Time is not meant to be plentiful. Make sure not to get bogged down on a single problem.

PROBLEM 1 (36 pts)

(a) Explain the difference between struct and union. Write a snippet of code where their difference in behavior/use is highlighted.

(b) Explain the meaning of a variable declared as int **x. Why is a variable y declared as a 2-D array int y[5][7] also of type int **y?

(c) printf() is a standard I/O function that takes a variable number of arguments. Why is it not necessary to declare printf() as int printf(int numargs, ...) where numargs specifies the number of arguments in a call to printf()? What is the function prototype of printf() based on how we have used it? What happens when calling printf("%d %d", x, y, z) with an extra argument, and why? Assume x, y, z are variables of type int.

PROBLEM 2 (36 pts)

(a) Suppose we have a structure defined as
typedef struct contacts {
    char *name; char *telnum; struct contacts *next; }
contacts_t;
which is used to store contact names and their telephone numbers in the first two members of the structure. In the third member, we store a pointer that contains the address of the next contact. Suppose s is declared as contacts_t *s. Assuming you have only two friends, "Alice" with phone number "765-000-0001" and "Bob" with number "765-000-0002", write a snippet of C code that uses malloc() and strcpy() to set up a linked list containing the two contacts.

(b) What is the difference between a function call returning to its caller and a process terminating by calling exit()? If exit() is called with argument 1, to whom is this value made known? What happens to a process when it calls execl()? Does this function call return to the caller? In what way is a system call (e.g., write()) different from a user library function (e.g., printf())?

(c) When a C program runs as a process, a stack frame is allocated (i.e., pushed) in the process’s stack area when a function call is made. The stack frame is deallocated (i.e., popped) when the function returns. What are the principal uses of the stack frame? Suppose a variable is declared as char *z within a function myfunc() that is called from main(). myfunc() makes a call to malloc(), z = (char *) malloc(sizeof(char)), before it returns. Where is z allocated in memory and how many bytes does it take up? Where in memory does malloc() allocate space pointed to by z? What happens to z when myfunc() returns? What do you think happens to the space allocated by malloc() when myfunc() returns? Discuss your reasoning.

PROBLEM 3 (28 pts)

(a) Suppose you are asked to write an app in C that reads from a file legacy executables, one to a line, preceded by a time value that specifies when in the future (in minutes), from now, to run the binary. For example,

15 /home/joe/a.out
90 /usr/bin/make

stored in file stuff-to-run.txt in the current directory of where your app is run specifies that the binary /home/joe/a.out is to be run after 15 minutes from now. /usr/bin/make is to be run 90 minutes from when the previous binary /home/joe/a.out is run. Using the concurrent server code for a basic shell as your starting point (the one discussed in class), sketch the changes you would make to arrive at a concurrent server code for the above C program.

(b) Running the program main() {float *r; r = 5.5; } we encounter a segmentation fault. Why is this the case? Suppose you want to inspect the value of r (an address) right after the segmentation fault occurs. Explain how you can accomplish this and write code that does it. You don’t need to worry about specifying the relevant header files.

BONUS PROBLEM (10 pts)
The main() function takes a variable number of arguments in the sense that we defined it using main(), main(int argc, char **argv), and main(int argc, char **argv, char **environ). This goes against the convention of defining functions with variable number of arguments such as int myprintf(int a, ...), where the first argument’s type must be fixed. Why do you think this is not a problem in the case of main() and the compiler is able to handle it?