

1. **Briefly** answer the following short-answer questions.

- a. Describe one benefit and one drawback of using a dynamically-linked (.so) library over using a statically-linked (.a) library.
- b. Describe how floating point numbers are stored. What are the fields and how are those fields utilized?
- c. For each of the following give value in the base indicated. You must assume all binary numbers are 8-bit 2's complement and all binary answers should be given in 8-bit 2's complement. You must assume all non-binary values are just representing a positive value in the base indicated.
 - i. 00101001_2 in decimal:
 - ii. $10001111_2 + 00001011_2$ in binary:
 - iii. 11011011_2 in decimal:
 - iv. AB_{16} in decimal:
 - v. $ABCD_{16}$ in binary:
 - vi. 174_{10} in hexadecimal:
- d. What is the difference between < and > when used on a UNIX command line?
- e. Explain the similarities and differences between the two function prototypes:

```
int func1(int* const a);  
int func2(const int* a);
```
- f. What does loop unrolling refer to? **Briefly** explain why it's advantageous.
- g. What Pthreads function can be used by a thread to obtain the return value from another thread? And what is the equivalent function for a process, instead of a thread?

2. Give the exact output that would be produced by the code below. You do not need to worry about the exact location of any whitespace characters. You only need to worry about exactly what text appears on which lines.

```
#include <stdio.h>
#define ARRSIZE 12

typedef struct{
    int size;
    int arr[ARRSIZE];
} SType;

int main() {
    char name[ARRSIZE]= "Jeff Jones";
    SType s1= {3, {7, 2, 4}};
    int *iptr;
    char *cptr;
    SType *sptr;

    iptr= s1.arr;
    printf("%d and %d\n", iptr[1], *iptr);
    cptr= &name[5];
    printf("%s\n", cptr);
    printf("%c and %c\n", *(cptr + 3), *(cptr - 4));
    cptr++;
    printf("%c and %c\n", name[3], cptr[3]);
    sptr= &s1;
    printf("%d and %d\n", sptr->size, *(sptr->arr + 2));

    return 0;
}
```

3. Give the output of the following program. Assume the program runs all of the way through without a fatal error or segmentation fault. If there are places that you do not know the actual value being printed, you should write "???" for the value printed. This uncertainty could occur because of uninitialized values being used or because of pointers into unallocated space.

```
#include <stdio.h>
#include <stdlib.h>

static int stint;

void funct(int a) {
    /* you don't know what happens in this function*/
    /* but no output takes place */
}

int main() {
    int a= 5, *b= &a, c[]={10, 21, 12};
    int *d, *e, f;

    printf("%d %d %d \n", *b, *c + 1, stint);
    e= b;
    b= malloc(sizeof(int));
    d= malloc(sizeof(int));
    *d= 7;
    *b= 9;
    *e= 11;
    printf("%d %d %d\n", *e, *d, a);
    funct(a);
    e= b;
    free(b);
    printf("%d %d %d\n", *e, f, a);

    return 0;
}
```

4. Suppose you love C so much you are writing your own C compiler. You've finished most of it, but you haven't written the `system()` standard library function yet. Below you will write a function that performs the functionality of the `system()` function. Your function will have the prototype:

```
void my_system(char *string);
```

Its parameter `string` represents a program which is to be executed, which may be followed by options or arguments, such as `"ls -l -a"` or `"cp file1 ../file2"`.

What your function will have to do is the following:

- It must create another process.
- The process created will have to break up the argument string into an array of strings. But you may assume that a function `parse()`, with the following prototype, already exists which does this:

```
void parse(char *string, char *strs[]);
```

Given a character string `string`, it breaks it up into separate strings (which were separated by whitespace) and produces an array of pointers to those strings in `strs`, with an additional pointer after the last string, with a NULL value.

- The created process will then have to execute the command that was in the argument string, by replacing itself with that program (and its arguments).
- After the command finishes executing, your function `my_system()` can then return.

The requirements regarding your function are as follows:

- Since your function is supposed to be implementing C's `system()` standard library function, assuming that it didn't exist, you **may not** use the standard library version of the function.
- You only need to write the function `my_system()`, not a complete source file. You may assume that any needed header files have already been included.
- You may assume that no command to be executed will ever have more than 40 arguments.
- Your function may assume that the command in its parameter `string` is always valid. If any other type of error occurs (such as for example a system call failing) it should print some type of message and cause the entire program to quit with an exit status of `-1`.

Comments are unnecessary, but your function must be written neatly, with good style and formatting.

You may want to use some of the following system calls, which are listed in no particular order:

```
int pipe(int filedes[2]);      int execvp(const char *file, char *const argv[]);
pid_t fork(void);             int execlp(const char *file, const char *arg, ...);
int close(int fd);            int dup2(int oldfd, int newfd);
pid_t wait(int *status);      pid_t waitpid(pid_t pid, int *status, int options);
```

5. Consider the following function in C:

```
void f(int *p, int n) {  
    printf("%d", p[n]);  
}
```

Give complete Y86 assembly code that a compiler could generate for the function (all the code for the entire function, including the function prologue and epilogue).

- You can assume the compiler doesn't perform any optimization.
- Recall that each data item (including integers and pointers) is 4 bytes on the Y86.
- Remember that the runtime stack grows downwards from address 0x1000.
- Note that array subscripting is being performed, but `p`, which is being used as an array, has no associated label, because it is a function parameter and not a global variable (parameters are stored in the runtime stack, while global variables are stored in the data area).
- Assume that the function will store its parameters' values in the runtime stack before doing any computation. It will store information in the runtime stack in the following order (from high to low addresses):

return address
p
n

6. Suppose you want to build a simple ATM application for depositing and withdrawing money safely from multiple ATMs simultaneously, and you want only one user/ATM to be updating a given account's balance at a time. Model each user/ATM as a thread, and write two functions, **deposit()** and **withdraw()**, each of which takes a struct of type **Account**, as declared below, and an unsigned integer amount as arguments, and updates the account balance in a thread-safe manner, returning the new account balance. If you need to, you can also write an initialization function with no arguments and no return value, called **init**, that initializes any variables needed to properly implement **deposit** and **withdraw**. You can assume that the initialization function would be called before any calls to either of the other functions.

The declaration for the struct and the prototypes for the functions are as follows:

```
typedef struct acc {
    unsigned int userid; /* unique ID */
    char *user_name;     /* pointer to string with user name */
    unsigned int balance;
} Account;

void init(void);
int deposit(Account *account, unsigned int amount);
int withdraw(Account *account, unsigned int amount);
```

You may want to use some of the following pthreads functions, which are listed in no particular order:

```
pthread_create(pthread_t *tid, pthread_attr_t *attr, void * (*func) (void *), void *arg)
pthread_join(pthread_t tid, void **val)
sem_init(sem_t *s, 0, unsigned int value)
sem_wait(sem_t *s)
sem_post(sem_t *s)
```