

## CMSC216: Practice Exam 3B SOLUTION

Spring 2026, University of Maryland

Exam period: 20 minutes

Points available: 40

**Problem 1 (15 pts):** Nearby is a description of the function `half_file()` and a `main()` which demonstrates its behavior. Complete this function in the space provided.

```

1 #include "half_file.h"
2
3 char *half_file(char *fname,
4                 size_t max_read_bytes,
5                 size_t *total_bytes_ptr);
6 // Reads the first half of bytes in 'fname' and
7 // returns a pointer to heap-allocated memory
8 // for it. Rounds down if the number of bytes is
9 // uneven. Reads 'max_read_bytes' each time
10 // input is done unless that would read past the
11 // halfway mark of the file. Sets argument
12 // 'total_bytes_ptr' to the number of bytes read
13 // which should be half the file size. Uses
14 // low-level UNIX I/O calls to determine file
15 // size and perform I/O operations. Prints and
16 // error and returns NULL if opening fails.
17
18 int main(int argc, char *argv[]){
19     size_t size; char *fname; char *contents;
20
21     fname = "input1.txt";
22     printf("input1.txt is:\n");
23     system("cat input1.txt");
24     contents = half_file(fname, 8, &size);
25     printf("half %s is %lu bytes:\n",fname,size);
26     printf("%s\n",contents);
27     free(contents);
28     // input1.txt is:
29     // 110 120 130 140 150 160 170 180
30     // half input1.txt is 16 bytes:
31     // 110 120 130 140
32
33     fname = "input2.txt";
34     printf("input2.txt is:\n");
35     system("cat input2.txt");
36     contents = half_file(fname, 5, &size);
37     printf("half %s is %lu bytes:\n",fname,size);
38     printf("%s\n",contents);
39     free(contents);
40     // input2.txt is:
41     // 110 120 130 140 150 160 170 180 190 200
42     // half input2.txt is 20 bytes:
43     // 110 120 130 140 150
44
45     return 0;
46 }

```

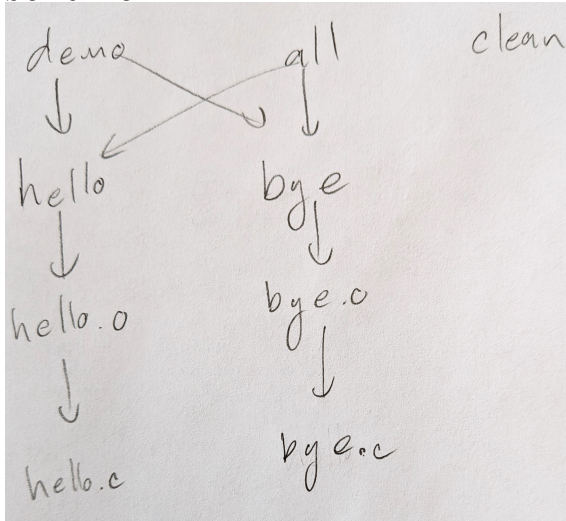
```

1 ////////////////////////////////////////////////// SOLUTION //////////////////////////////////////
2 #include "half_file.h"
3 char *half_file(char *fname,
4                 size_t max_read_bytes,
5                 size_t *total_bytes_ptr)
6 {
7     struct stat sb;
8     int result = stat(fname, &sb);
9     if(result==-1){
10        perror("Couldn't stat file");
11        return NULL;
12    }
13    int fd = open(fname, O_RDONLY);
14    if(fd == -1){
15        perror("Couldn't open file");
16        return NULL;
17    }
18    size_t total_bytes = sb.st_size / 2;
19    char *contents = malloc(total_bytes + 1);
20    size_t offset = 0;
21    while(offset < total_bytes){
22        size_t read_count = max_read_bytes;
23        if(total_bytes-offset < max_read_bytes){
24            read_count = total_bytes-offset;
25        }
26        size_t n = read(fd, contents+offset, read_count);
27        if(n != read_count){ // optional error check
28            printf("unexpected number of bytes read:\n");
29            printf("expect: %lu\n",read_count);
30            printf("actual: %lu\n",n);
31        }
32        offset += n;
33    }
34    close(fd);
35    contents[total_bytes] = '\0';
36    *total_bytes_ptr = total_bytes;
37    return contents;
38 }

```

**Problem 2 (10 pts):** Nearby is a sample Makefile / Bakefile. In the space below, draw a picture of the DAG that would be created from this file showing the dependencies between the targets in it.

*SOLUTION:*



```

1 demo : hello bye
2     @ echo Running programs
3     ./hello
4     ./bye
5     @ echo Done running programs
6
7 all : hello bye
8
9 hello : hello.o
10     gcc -o hello hello.o
11     echo Ready to say hello
12
13 hello.o : hello.c
14     gcc -c hello.c
15
16 bye : bye.o
17     gcc -o bye bye.o
18
19 bye.o : bye.c
20     gcc -c bye.c
21
22 clean :
23     rm -f hello.o bye.o hello bye
  
```

**Problem 3 (10 pts):** If the command `bake all` were run in the folder with the above Bakefile, what commands would be run to complete the build? Show any valid ordering for the commands.

*SOLUTION:* One valid ordering is the following commands. Comments indicate the targets associated with the commands and are not necessary for credit.

```

gcc -c hello.c           # for hello.o

gcc -o hello hello.o    # for hello
echo Ready to say hello

gcc -c bye.c            # for bye.o

gcc -o bye bye.o        # for bye

                        # finished all
  
```

**Problem 4 (5 pts):** Describe the difference between “Normal” and “Abnormal” process exits. How does a parent process determine if a child process exits normally after successfully completing, exits normally on encountering an error, or exits abnormally? Give code snippets demonstrating how these cases are detected.

*SOLUTION:* The `wait()` / `waitpid()` system call allows a parent to receive the Status of a child process call then determine whether a Normal or Abnormal exit occurred via the macros `WIFEXITED()` and `WIFSIGNALED()` respectively. On a normal exit, success from child a child process is usually indicated by a 0 exit code while a Non-zero exit code is some sort of error.

```

1 { int status; // PARENT CODE
2   wait(&status); // Set child status
3   if(WIFEXITED(status)){ // Normal Exit
4     int exit_code = WEXITSTATUS(status);
5     if(exit_code == 0){ ...; } // Success
6     else { ...; } // Errors
7   }
8   else if(WIFSIGNALED(status)){...;} // Abnormal Exit
9 }
  
```