Question 1: (10 points) What is the name of the (user space) wrapper function of the system call for the shutdown2 command (2 points)? Where is this wrapper function invoked (3 points)? Where is this function defined (5 points)?

- 1.1: the name of the wrapper function is shutdown2
- 1.2: this function is invoked in user-space code, main() in shutdown.c in this case
- 1.3: this function is defined in user.h

Question 2: (10 points) Explain (with the actual code) how the above wrapper function triggers the system call.

1. `syscall.h`: #define SYS_shutdown2 24
   This file contains macro definitions for syscall numbers. In this case, `SYS_shutdown2` is defined as 24. This macro provides a symbolic name for the syscall number associated with **shutdown2**. This makes it easier to remember and use in the code compared to the raw syscall number

2. `user.h`: int shutdown2(char* msg);
   This file contains function prototypes for user-level functions. The declaration `int shutdown2(char* msg)`; indicates the function prototype for the **shutdown2** function

3. `syscall.c`: extern int sys_shutdown2(void);
   This file contains an array of function pointers, each corresponding to a specific syscall number. The line `extern int sys_shutdown2(void);` declares the `sys_shutdown2` function, which is the kernel-level implementation of the **shutdown2** syscall

3. `syscall.c` (continued): [SYS_shutdown2] sys_shutdown2,
   The line `[SYS_shutdown2] sys_shutdown2` associates the syscall number `SYS_shutdown2` with the `sys_shutdown2` function pointer. This association allows the syscall dispatcher to route the **shutdown2** syscall to the appropriate kernel-level function

4. `sysfile.c`: int sys_mkdir2(void){
   This file contains the implementations of various file-related syscalls. The `sys_shutdown2` function is an example of a kernel-level implementation of the **shutdown2** syscall. This function performs the actual work of creating a directory in the filesystem

5. `shutdown2.c`: shutdown2(msg);
   This file contains user-level code that invokes the **shutdown2** syscall. The main function is the entry point of the program. Inside main is the code that calls the **shutdown2** function with appropriate arguments. This code is responsible for initiating the **shutdown2** syscall from user space

Question 3: (10 points) Explain (with the actual code) how the OS kernel locates a system call and calls it.

1.The OS kernel locates a system call by firstly invoking it through a library function, which eventually triggers a software interrupt. In this case, it's called the 'exit' system call.

```
int
main(int argc, char * argv[])
{
    int i = atoi(argv[1]);

    // "msg" now holds the shutdown message provided by the user
    //shutdown2(msg);

    if(i<1 || i>3)
        printf(1,"Available options for uptime output: \n[\n1=seconds,\n2=days,\n3=years\n]");

    uint ut = uptime2(i);

    if(i == 1){
        printf(1, "Current Uptime in ticks: %d", ut);
    } else if(i == 2){
        printf(1, "Current Uptime in seconds: %d", ut);
    } else if(i == 3){
        printf(1, "Current Uptime in minutes: %d", ut);
    }

    exit(); //return 0;
}
```

2.The ‘exit’ function invokes the software interrupt which is done in the file ‘usys.S’

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
```

3.When the interrupt is triggered, the CPU switches to kernel mode and jumps to a location within the IDT(interrupt descriptor table which is done in ‘trapasm.S’ and ‘trap.c’. The handler determines that a system call interrupt is called and then calls ‘syscall()’ to handle the interrupt.

4. The syscall function in ‘syscall.c’ reads the system call number from the ‘%eax register” and uses it to index into a table of call handler functions.

```
144  syscall(void)
145  {
146    int num;
147    struct proc *curproc = myproc();
148
149    num = curproc->tf->eax;
150    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
151      curproc->tf->eax = syscalls[num]();
152    } else {
153      cprintf("%d %s: unknown sys call %d\n",
154              curproc->pid, curproc->name, num);
155      curproc->tf->eax = -1;
156    }
157  }
158
```

5.Then the actual system call handler function is executed.

```
126  int
127  sys_smile(int num)
128  {
129    if (argint(0, &num) < 0)
130      return -1;
131
132    for (int i = 0; i < num; i++)
133    {
134      cprintf("smile %d: (◕ ‿ ◕)\n", i + 1);
135    }
136    return 0;
137  }
138
```

Click to collapse the range.

6. After the system call handler completes, control is returned back to the user space to continue execution.

Question 4: (10 points) How are arguments of a system call passed from user space to OS kernel?

System call arguments can be passed in 3 ways: in registers, onto a stack, and in a block in a heap. Arguments that are placed or pushed onto the stack by the process and popped off the stack by OS kernel code. Typically, this is done with functions like argstr() for strings and argint() for integers. The user is automatically low level, but when a

user runs a command that invokes a system call there is a transition to kernel mode. System call arguments involve a “trap” instruction that raises the privilege level and jumps into the kernel mode. Happens anytime a user tries to access something they shouldn’t, but sys calls have a unique number so when it reaches the kernel, it returns the call, and knows what the call is.