1. Where is the wrapper function of this system call that obtains the invocation times (i.e., the int get_syscall_count(int call type) function) defined?

   The wrapper function is defined in user.h file. All the wrapper functions for system calls are defined in this file. Test program in our case need to import user.h to be able to call the wrapper function.

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

   Steps involved in triggering a system call:

   a) The test program invokes system call by calling wrapper function (get_syscall_count(int call type) defined in user.h.

```
int get_syscall_count(int);
int reset_syscall_count(void);
```

   b) A macro defined in usys.S is used to call a system call.

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


SYSCALL(get_syscall_count)
SYSCALL(reset_syscall_count)
```

   The '\' in C means connect to the next line. The code defines name of the system call as global in the first line, then copies the number associated with system call to register value %eax. System call number will be defined in syscall.h file as shown below. $T_SYSCALL is a constant defined in traps.h. It helps the kernel to understand that it's a system call. It is same for all the system calls in an OS. The value is equal to 64 in the xv6 implementation. The last line returns from this function.

   Syscall.h (stores number associated with system call)

```
#define SYS_get_syscall_count 23
#define SYS_reset_syscall_count 24
```

Hence SYSCALL(get_syscall_count) can be expanded as
#define SYSCALL(get_syscall_count) .globl get_syscall_count; get_syscall_count: movl 23, %eax; int 64; ret

c) Then the wrapper function calls a trap machine instruction, which causes the processor to switch from user mode to kernel mode and execute system call.

```
Void trap(struct trapframe *tf){
        if(tf->trapno == T_SYSCALL){
        if(myproc()->killed)
                exit();
        myproc()->tf = tf;
        syscall();
        if(myproc()->killed)
        exit();
        return;
        }
```

d) In response to the trap, kernel calls syscall() routine in syscall.c. Invokes appropriate system call service routine which is found using the system call number to index a table of system call service routines. We need to define our system call in system call routine table.

```
static int (*syscalls[])(void) = {
        .
        .
        .

        [SYS_get_syscall_count] sys_get_syscall_count,
        [SYS_reset_syscall_count] sys_reset_syscall_count,
        };
```

The below lines in syscall.c says that system call is found somewhere in other file.

```
        extern int sys_get_syscall_count(void);
        extern int sys_reset_syscall_count(void);
```

e) Finally, the kernel calls the system call defined in sysproc.c.

```
int sys_get_syscall_count(int type){
        argint(0,&type);
        if(type==0){
                return forkcount;
```

```
 }
else if(type==1){
      return exitcount;
}
else if(type==2){
      return waitcount;
}
 else{
      return -1;
}
}

int sys_reset_syscall_count(void){
      forkcount = 0;
      waitcount = 0;
      exitcount = 0;
return 0;
}
```

3. Explain (with the actual code) how the OS kernel locates a system call and calls it (i.e., the kernel-level operations of calling a system call).

   In user mode, OS copies the value associated to the system call in %eax register and causes a trap. In response to the trap, the kernel switches from user mode to kernel mode, saves the register value to kernel stack and invokes syscall function in syscall.c. Below is the part of trap function which invokes syscall().

```
Void trap(struct trapframe *tf){
      if(tf->trapno == T_SYSCALL){
      if(myproc()->killed)
            exit();
      myproc()->tf = tf;
      syscall();
      if(myproc()->killed)
      exit();
      return;
      }
```

   In the syscall function, kernel gets system call number from the register and uses that number to index a table of system call service routines(syscalls). In the below code snippet kernel is getting system call number and storing it in num variable. It validates if num is a valid system call number and then stores the address of system call in the register %eax.

```
Void syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

Syscalls table: It is used to locate system calls.

```
static int (*syscalls[])(void) = {

       .
       .
       .
             [SYS_close]    sys_close,
       [SYS_shutdown]      sys_shutdown,
       [SYS_get_syscall_count] sys_get_syscall_count,
       [SYS_reset_syscall_count] sys_reset_syscall_count,
       };
```

As we can see the syscalls table is a list of function pointers where each pointer, points to a system call implementation which acts as an interrupt service routine. Finally, the kernel gets the address of the system call and calls it.

4. How are arguments of a system call passed from user space to OS kernel?

   The OS pushes the arguments to the stack in user space. The register value %esp points to the top of the stack. The saved user %esp points to a saved program counter, and then the first argument. Argint is a function used to get the arguments from the stack in kernel space. Below is the implementation of the argint function where n is the argument number. If it is first argument, n=0 and it will store the address %esp+4 in a pointer ip.

```
int
argint(int n, int *ip)
{
  return fetchint((myproc()->tf->esp) + 4 + 4*n, ip);
}
```

We can use the argint function to fetch the arguments as seen below.

```
int sys_get_syscall_count(int type){
    argint(0,&type); // stores value of the argument in variable called type.
    if(type==0){
      return forkcount;
    }
    else if(type==1){
      return exitcount;
    }
    else if(type==2){
      return waitcount;
    }
    else{
      return -1;
    }
}
```