OS @ run (kernel mode)	Hardware	Program (user mode)

Program (user mode)

Within the user program:

```
...
close(fd)
```

A system call is made with a bad `fd`

Within `usys.S`:

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

The macro definition causes `name` to be populated with `close` so a global close is defined which gets called by the user process. This causes `close` to move the `SYS_close` number (which is defined by a macro in `syscall.h`) into the `%eax` register. Then a call to an interrupt using `int` with `T_SYSCALL` (which is 64) is made.

Hardware

Within `vectors.S`:

```
.globl vector64
vector64:
  pushl $0
  pushl $64
  jmp alltraps
```

At global vector64, the instructions are to push 0 and the trap number 64 (to kernel stack), move to kernel mode, and jump to `alltraps`.

Within `trapasm.S`:

```
.globl alltraps
alltraps:
  # Build trap frame.
  ...
  # Set up data segments.
  ...
  # Call trap(tf), where
tf=%esp
```

The trap frame is built, the data segments are set up, and `trap()` is called with `%esp` holding 64.

OS @ run (kernel mode)

Within `trap.c`:

```
void trap(struct
trapframe *tf)
{
  if(tf->trapno ==
T_SYSCALL){
    ...
    myproc()->tf = tf;
    syscall();
```

The trap handler enters the if statement for system calls and `syscall()` gets executed.

Within `syscall.c`:

```
static int
```

```
(*syscalls[])(void) = {
...
[SYS_close]   sys_close,
};
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]();
  }
```

The `eax` reg value (which holds the system call number for `sys_close`) is used to call `sys_close()` through the array `syscalls[num]()` and puts the result value into `eax` register

Within `sysfile.c`:

```
int sys_close(void)
{
  int fd;
  struct file *f;
  if(argfd(0, &fd, &f) <
0)
    return -1;
  ...
  fileclose(f);
```

Finally, `sys_close()` gets called which then calls argfd() to interpret the nth argument as a file descriptor.

Within `file.c`:

```
void
fileclose(struct file
*f)
{
  struct file ff;

  acquire(&ftable.lock);
  if(f->ref < 1)
    panic("fileclose");
```

The file table is locked and the if statement is entered checking if there are any references to that file. Because it is faulty there are none and it enters `panic()`.

Within `console.c`:

```
void panic(char *s)
{
  ...
  cprintf(s);
  cprintf("\n");
```

Feedback is displayed to the user.

How to use `countTraps()`:

The system call `countTraps(int c)` (where `int c` represents a system call number) takes in one parameter which is a system call number. To use `countTraps()`, simply include the `user.h` header file.

This version of xv6 has a total of 22 system calls. If the user process calls `countTraps()` with a system call number between 1 and 22, it will return the number of times that specified system call has been made within that specific user process.

If the user process calls `countTraps` with 0 as the parameter, `countTraps` will return the total number of system calls made by that user process. It will also print out the number of traps the entire system has made up until that point (which includes the total system calls made, hardware interrupts, and software interrupts).

If an invalid parameter is passed or no system calls have been made, -1 is returned.

To make it easier for the user, they can include the `syscall.h` header file within their program. The `syscall.h` header file contains textual substitutions (macros) that define each system call with its call number. For example, if the user wished to obtain the count of `fork()` system calls made by that process, they can call `countTraps(SYS_fork).`

Design of `countTraps()`:

The design of the system call `countTraps()` is made possible by keeping track of the number of each individual system call type made by a given user process through an array contained in the metadata of each process. It is also implemented by keeping track of the total number of traps (which includes system calls, hardware interrupts, and software interrupts) of the entire system. Given a system call number (or its position in the system call vector), `countTraps()` gets the current user process's metadata (`proc struct`) by calling `myproc()`, accesses the array `int[] sys_arr` (which contains all the counts of each system call type), and indexes the array to access the entry relative to that system call number.

If the given system call number is 0, `countTraps()` returns the total system calls of the user process and prints the trap counts of the entire system. The reason 0 was chosen for this specific return is because the system call numbers begin at 1.

The total count of hardware and software interrupts per process (not across the system) is not kept track of. The reason for this design choice is that most hardware and software interrupts do not make it possible to access that specific process's metadata through calling `myproc()` mainly because these interrupts result in the process actually being killed. This version of xv6 also does not support any keyboard interrupts that could temporarily suspend a process.

How `countTraps()` was implemented:

To implement the system call `countTraps(int c)` additions had to made to the following source code files:

- `proc.h` - An `int[] sys_arr` array and `int sysc` declaration were added to `struct proc` to keep track of the count of each process's total system calls as well as the count of each individual system call type for each process.
- `proc.c` - Code was added to zero out the `int[] sys_arr` array and set `int sysc` to 0 within both `userinit()` (where the first process is made) and the child process in `fork()`. The reason I chose to reset the counts of the child process once forked is because I figured that even though the child process inherits most of the parent's attributes, it does not exist before it is forked and therefore its counts should start at 0.
- `syscall.h` - A line of code has to be added to define a macro for the system call number or its position in the system call vector
- `defs.h` - A line of code had to be added for a forward declaration of the `int countTraps(int)` system call.
- `user.h` - A line of code had to be added to define the `countTraps(int)` system call that can be called through the shell.
- `sysproc.c` - The actual body of the `countTraps(int)` system call was added here. It also contains three `int` variables (`total_sysc`, `total_swic`, and `total_hwic`) that hold the total system call, total hardware interrupt, and total software interrupt counts that occurred over the entire system. A line of code was also added to each system call to increment the correct entry of array `int[] sys_arr` (which is located inside the `struct proc` of `proc.h`).
- `syscall.c` - Code was added to externally define the function that connects the shell and kernel. The macro defined in `syscall.h` was used here to add it to the system call vector.
- `usys.S` - A line of code was added to define a macro to connect the user call to the actual system call `sys_countTrap()`.
- `syscfile.c` - A line of code was added to each file system call to increment the correct entry of array `int[] sys_arr` (which is located inside the `struct proc` of `proc.h`).
- `trap.c` - Code was added to the `trap()` function that increments the total system call, total hardware interrupt, and total software interrupt counts of the system. The trapframe maps number 0 to 31 as software interrupts/exceptions, 32 to 63 as hardware interrupts, and 64 as system call traps. The `trap()` function begins by checking if the trap number maps to system calls, then checks if it maps to numerous hardware interrupts, and finally (if all else is false), it assumes that it must have been a software interrupt such as a fault in user space like dividing by 0 or an issue in kernel space. Thus, the trap counts (`total_sysc`, `total_swic`, and `total_hwic`) just have to be incremented based on which trap had occurred.

Test Cases:

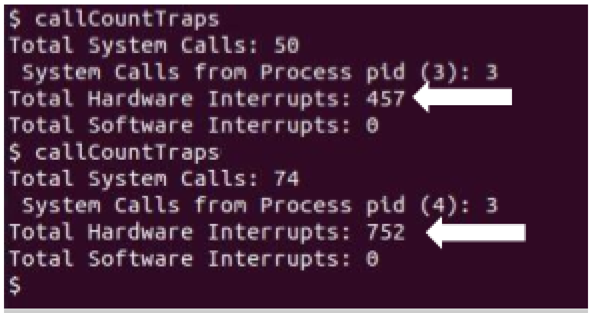
Equivalence Partitioning			
`int c == 0` The given parameter is 0 so the total traps of the system should be printed and the total system calls of the calling process is returned		`int c >= 1 && int c <= 22` The given system call number falls into bounds for a specified system call.	`int c < 1 || int c > 22` and `int c != 0` The given system call number is out of bounds.

Majority of the **hardware interrupts** are timer interrupts
 	
`callCountTraps`
//wait 1 second
`callCountTraps`
 	
Verses
 	
`callCountTraps`
//wait 10 seconds
`callCountTraps`
	Divide by zero causing **software interrupt**
	
	`divideByZero`
	`callCountTraps`
	`divideByZero;`
	`callCountTraps`
		Parents and child processes make numerous different system calls:
		
		In `parentAndChild.c`:
		Parent process makes 3 bogus close() calls then forks
		
		Child makes 1 bogus close() call then exits
		
		Parent waits for process
			The exception occurring in `badCallCountTraps.c` should result in `countTraps()` returning -1.

Test 1	Test 2	Test 3	Test 4

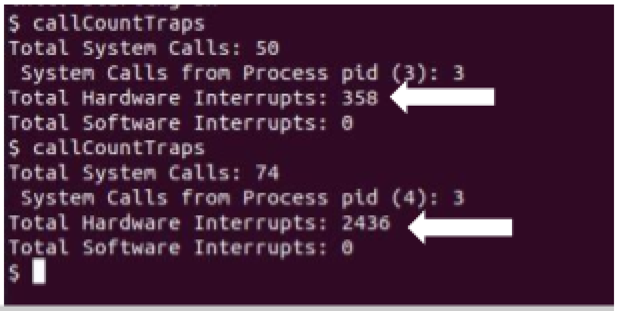
To test `countTraps()`, three possible groupings of parameters are partitioned. The first being that 0 was passed, the second being that a specification for a particular system call was passed, and the last being that the parameter is out of bounds. To demonstrate these partitionings, four different user programs were made:

- `divideByZero.c`: Simply performs an erroneous divide by zero that will fault.
- `callCountTraps.c`: A single call is made to `countTraps(0)` so the system's entire trap counts are printed.
- `parentAndChild.c`: A parent process makes 3 bogus calls to `close()`, 1 `fork()` call, and 1 `wait()` call and the child process makes only 1 bogus call to `close()`. `countTraps()` is called after every system call to show the system call counts of the parent and child processes.
- `badCallCountTraps.c`: A single call is made to `countTraps(30)` with 30 being out of bounds.



```
$ callCountTraps
Total System Calls: 50
 System Calls from Process pid (3): 3
Total Hardware Interrupts: 457
Total Software Interrupts: 0
$ callCountTraps
Total System Calls: 74
 System Calls from Process pid (4): 3
Total Hardware Interrupts: 752
Total Software Interrupts: 0
$
```

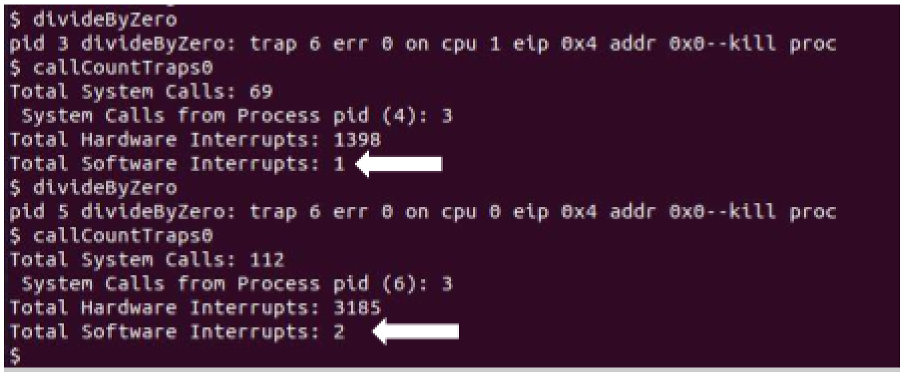
Test 1A



```
$ callCountTraps
Total System Calls: 50
 System Calls from Process pid (3): 3
Total Hardware Interrupts: 358
Total Software Interrupts: 0
$ callCountTraps
Total System Calls: 74
 System Calls from Process pid (4): 3
Total Hardware Interrupts: 2436
Total Software Interrupts: 0
$
```

Test 1B

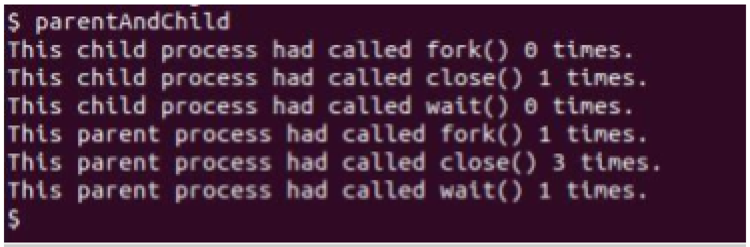
In test 1, we run `callCountTraps` twice, which displays the total traps that occurred between two different moments. In test 1A, we wait about a second before running `callCountTraps` a second time. The difference in hardware interrupts is 752 - 457 = 295. In test 1B we wait about 10 seconds before running `callCountTraps` a second time. The difference in hardware interrupts is now 2436 - 358 = 2078. Test 1B has about twice the difference than test 1A which makes sense as the majority of the hardware interrupts are timer interrupts which should be produced by the clock chip at a pretty constant rate. Because we waited about 10 times longer in test 1B, the difference was about 10 times greater. Thus `callCountTraps` should be counting all or at least the majority of the hardware interrupts.



```
$ divideByZero
pid 3 divideByZero: trap 6 err 0 on cpu 1 eip 0x4 addr 0x0--kill proc
$ callCountTraps0
Total System Calls: 69
 System Calls from Process pid (4): 3
Total Hardware Interrupts: 1398
Total Software Interrupts: 1
$ divideByZero
pid 5 divideByZero: trap 6 err 0 on cpu 0 eip 0x4 addr 0x0--kill proc
$ callCountTraps0
Total System Calls: 112
 System Calls from Process pid (6): 3
Total Hardware Interrupts: 3185
Total Software Interrupts: 2
$
```

Test 2

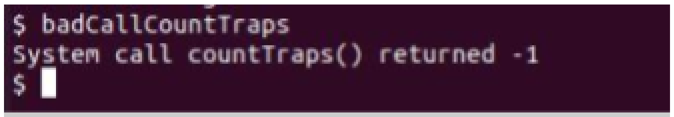
In test 2, we run `divideByZero` then `callCountTraps` to show that `countTraps()` is displaying the correct software interrupt counts. The software interrupts go from 1 to 2 as `divideByZero` was run twice.



```
$ parentAndChild
This child process had called fork() 0 times.
This child process had called close() 1 times.
This child process had called wait() 0 times.
This parent process had called fork() 1 times.
This parent process had called close() 3 times.
This parent process had called wait() 1 times.
$
```

Test 3

In test 3, we run `parentAndChild` to show that `countTraps()` returns the correct counts.



```
$ badCallCountTraps
System call countTraps() returned -1
$
```

Test 4

In test 4, we run `badCallCountTraps` to show that `countTraps()` returns -1 when faulty parameters are given.