XV6 ASSIGNMENT REPORT

ABHAY SHANKAR K: CS21BTECH11001

1. Inner workings of System Calls

1.1. **syscall.h.** `syscall.h` contains several symbolic constants with the prefix `SYS_`, and these numbers represent the various system calls implemented in xv6. When a system call occurs, the value of register `eax` is set to this number.

1.2. **syscall.c.**

- `syscall.c` contains the `syscall()` function, which reads the register `eax` and calls the appropriate system call.
- Our modifications also elude the name of the system call and print it to console, all save the `write()` call, which is the selected victim of the allowed exclusion of one system call. This is due to the abundance of write calls - if each call were logged, the relevant output cannot be seen.
- It also contains functions to extract the arguments provided to the system calls, viz. `argint()`, `argptr()`, etc. These functions are called within the definitions of the system calls in kernel mode, in lieu of passing the arguments as parameters.

1.3. **sysproc.c, sysfile.c.** These files contain the definitions of the system calls, i.e. functions with the prefix `sys_`. This is the business logic of each system call that executes after the interrupt is raised.

Specifically, sysfile.c contains system calls pertaining to various filesystem actions such as `sys_read()`, `sys_write()`, `sys_open()`, etc.

1.4. **usys.S.** This file contains the `SYSCALL` macro, using which the System Call API for user programs are defined. It does the following:

(1) Sets the eax register to the system call number.
(2) Raises interrupt with code 64 (`T_SYSCALL`, the interrupt representing a system call)

1.5. **user.h.** This file contains the declaration of the System Call API (These functions are later defined, in assembly language, in `usys.S`).

1.6. **Flow.** :

Consider a `date()` call, which we have implemented. This call sets `eax` to 22 (`SYS_date` = 22), and raises the System Call interrupt (`T_SYSCALL` = 64).

The handler for Interrupt 64 is the `syscall()` function, which reads `eax`, and calls the appropriate function (basically, a system call backend). In this case, the function is `sys_date()`.

The Interrupt Handler itself is fairly convoluted, defined in `trap.c` and `trapasm.S`. This eventually calls `syscall()`.

The business logic within the `sys_date()` function is fairly straightforward:

- Initially, we extract the argument to the `date()` call, which is a pointer to a struct (which is on the stack). This is extracted as an int for simplicity, then cast to the appropriate type before being passed as an argument to `cmostime()`.
- Most of the work is done by the `cmostime()` function, which initialises the fields of an instance of `struct rtcdate` given a pointer to it. It calculates these values from the system clock and assigns them to the apropriate fields.
- The contents are printed to console in the `main()` function of `mydate.c`

2. Page-table experimentation

(1) Without any arrays, the program occupies two pages. The size of the text segment is unknown, but due to the fixed stack size, there will be some lower bound on the number of pages allocated.

(2) The global array occupies ten pages (∼40kb in array, 4kb per page), so the process occupies twelve pages after program modification.

(3) The local array causes the program to trap and exit - specifically, trap 14, which is a page fault. This is because the size of the stack is fixed, and cannot contain 40kb.

(4) Using malloc soves this, since the heap is expandable, and the process occupies 12 pages.

(5) The lines for all the above situations are present in the source code, but have been commented. The program as submitted follows the last scenario.

```
Entry number: 0 | Physical address: dee2000 | Virtual address: 0
Entry number: 1 | Physical address: dedf000 | Virtual address: 2000
Entry number: 2 | Physical address: dfbc000 | Virtual address: 3000
Entry number: 3 | Physical address: df76000 | Virtual address: 4000
Entry number: 4 | Physical address: dfbf000 | Virtual address: 5000
Entry number: 5 | Physical address: dfc0000 | Virtual address: 6000
Entry number: 6 | Physical address: dfc1000 | Virtual address: 7000
Entry number: 7 | Physical address: dfc2000 | Virtual address: 8000
Entry number: 8 | Physical address: dfc3000 | Virtual address: 9000
Entry number: 9 | Physical address: dfc4000 | Virtual address: a000
Entry number: 10 | Physical address: dfc5000 | Virtual address: b000
Entry number: 11 | Physical address: dfc6000 | Virtual address: c000
```

```
sh: fork->1->3
sh: sbrk->12->16384
mypgtPrint: exec->7->0
pid 3 mypgtPrint: trap 14 err 5 on cpu 0 eip 0x21 addr 0x1fcc--kill proc
sh: wait->3->3
```

- The virtual addresses remain the same for each execution, but the physical addresses change.
- This is because the virtual addresses are generated by a compiler, whereas the physical addresses are allocated by the MMU.
- It is observed that pages are not always allocated in contiguous order, which is the entire purpose of paging.
- However, significant portions are contiguous, indicating that there is less load on the memory - which is true, since there is only one process running, and one process (the shell) in the background, waiting.