CS 344 Assignment 2A Report

Group No 6:

Aditya Trivedi 190101005
Akshat Arun 190101007
Atharva Vijay Varde 190101018
Shashwat Sharma 190123055

Applying changes and executing:

1) Using Patch file:
    - Copy the file patch.txt into “xv6-public” folder
    - `cd xv6_public`
    - `git apply patch.txt`
    - `make clean && make qemu`
2) Using modified files:
    - Copy the files in the “modified files” folder and paste/replace into “xv6-public” folder.
    - `make clean &&  make qemu`

Task 1:
The Execution Flow

- The interrupts generated by pressing a key on the keyboard are handled by `kdbintr()`, which ultimately calls `consoleintr()` with a pointer to function `kdbgetc()` which gets key input. `kdbgetc()` generates an integer value based on the pressed key as according to the mapping defined in kbd.h.

- `consoleintr()` is responsible for interpreting these integer codes, and manipulating the internal input buffer. It has special cases for Ctrl key combinations, backspaces etc. To manipulate the on-screen buffer, it calls `consputc()`, which calls `uartputc()` for handling input to the host terminal, and `cgaputc()` for the qemu display.

- Finally, on pressing enter, the `consoleread()` function is woken up by a call to `wakeup(&input.r)`. This function reads the input buffer from the index `input.r` to `input.w-1` (inclusive), and passes this string to the shell. Our command is thus successfully read into the shell for execution.

It is to be noted that for any input, we must manipulate two buffers, the internal input buffer for the kernel, and the on-screen buffer used by the cga display. Hence, we shall modify two functions, `consoleintr` and `cgaputc` respectively.

Task 1.1 Caret Navigation

Currently, the left and right arrow keys get interpreted as characters due to the default case, and their integer codes are displayed as characters according to the CGA character mapping table. Moreover, we will need to shift the characters in the buffers appropriately when inserting/deleting characters. Hence, we do the following modifications:

1) For the left arrow key, we first add a case so that it is not interpreted as a character to be displayed. Next, we decrement `input.e`, which tracks the position of the cursor inside the internal buffer. Finally, we call `consputc` with the value LEFTARR.

2) Inside `cgaputc` we add a condition to check for LEFTARR, which if true, we decrement the `pos` variable, which is used to keep track of the position of the on-screen cursor. The right arrow key is handled similarly, with increments instead of decrements.

Code:

```
case 0xe4:  // LEFT ARROW
    if (input.e != input.r) {
        input.e--;
        consputc(LEFTARR);
    }
    break;
case 0xe5:  // RIGHT ARROW
    if (input.e < input.w) {
        input.e++;
        consputc(RIGHTARR);
    }
    break;
```

3) For handling the backspace key and normal character input, The internal buffer is first shifted and/or character is inserted appropriately. We then update the screen in four steps.

4) First, the cursor is shifted to the rightmost point of the input using `consputc(RIGHTARR)`. Next, we clear the input till the previous position of the cursor. We then insert the characters from the updated internal buffer. Finally, we use `consputc(LEFTARR)` to shift the cursor back to its correct location.

Code :

```
case C('H'):
case '\x7f':  // Backspace
    if (input.e != input.r && input.e > 0) {
        for (uint i = input.e; i < input.w; ++i) {
            input.buf[(i - 1 + INPUT_BUF) % INPUT_BUF] =
                input.buf[i % INPUT_BUF];
        }
        for (uint i = input.e; i < input.w; ++i) {
            consputc(RIGHTARR);
        }
        for (uint i = input.e - 1; i < input.w; ++i) {
            consputc(BACKSPACE);
        }
        for (uint i = input.e - 1; i + 1 < input.w; ++i) {
            consputc(input.buf[i % INPUT_BUF]);
        }
```

```
                for (uint i = input.e; i < input.w; ++i) {
                    consputc(LEFTARR);
                }
                input.e--;
                input.w--;
            }
            break;
```

Task 1.2 Shell History

1) The up and down arrow keys currently, if typed, print the ascii values in the console, we have implemented the shell history feature by editing console.c and creating a history system call. The up and down arrows now retrieve the next and last item in history respectively.

   To do so, firstly we define the maximum history we can accommodate (16, according to question) and the maximum length each command can have (128) in history.h file. This file will be included wherever we need access to these values.

   Code :

```
#define MAX_HISTORY (16)
#define MAX_BUFFER_LEN (128)
```

   We use a history buffer along with 2 counters - total history counter and current history counter which help keep track of whether the buffer is full and which command is to be displayed.

   Code :

```
char history_buffer[MAX_HISTORY][MAX_BUFFER_LEN];
int histCnt = 0;
int currCnt = 0;
```

   In console.c in the `consoleintr` function we add a case for up arrow, down arrow and enter case.

   - In the up arrow case, we first move the cursor rightwards and clear the input, after which we retrieve the appropriate command from the history buffer. This is done by **decrementing** the current history counter which keeps track of how deep in the history buffer we are. We then print this to the console.

   - In the down arrow case, we first move the cursor rightwards and clear the input, after which we retrieve the appropriate command from the history buffer. This is done by **incrementing** the current history counter which keeps track of how deep in the history buffer we are. We then print this to the console.

- In enter case, we add the command entered by user to our history buffer, this is done using the `add_to_history` function defined in console.c

  Code :

```
void add_to_history(char *new_command) {
   int unit_size = sizeof(char);

   int null_command = (new_command[0] == NULLCHAR);

   // if it is a null command then it is not added to the buffer
   if (null_command) return;

   int actualLen = strlen(new_command);
   int length = actualLen;
   if (MAX_BUFFER_LEN - 1 < actualLen) {
       length = MAX_BUFFER_LEN - 1;
   }
   // will be used to check if history buffer has enough space
   int histLeft = histCnt < MAX_HISTORY;

   if (histLeft) {
       // can still accommodate new entry
       histCnt++;
   } else {
       // need to shift all entries
       for (int i = 0; i + 1 < MAX_HISTORY; i++) {
           memmove(history_buffer[i], history_buffer[i + 1],
                   unit_size * length);
       }
   }
   // adding new command to existing history buffer
   memmove(history_buffer[histCnt - 1], new_command, unit_size *
length);
   history_buffer[histCnt - 1][length] = NULLCHAR;

   // updating the current counter value
   currCnt = histCnt;
   return;
}
```

2) For adding history system call, we modify the following files (just like assignment 1)
   a) syscall.h
   b) syscall.c
   c) user.h
   d) usys.S

   Now we need to implement the “history” command in the shell user program. We do this by editing the sh.c file. In the main() function we just check if the entered command is “history” if it is then we initialise a local buffer and store commands using history system call. We keep collecting history till a non zero value is returned.

Code :

```
else if(buf[0] == 'h' && buf[1] == 'i' && buf[2] == 's' && buf[3] ==
't' && buf[4] == 'o' && buf[5] == 'r' && buf[6] == 'y'){
     buf[strlen(buf)-1] = 0;
     int success_flag = 0;
     int unit_size = sizeof(char);
     char buffer[MAX_HISTORY][MAX_BUFFER_LEN];
     int i = 0;
     printf(1, "*************************************\n");
     printf(1, "               HISTORY              \n");
     printf(1, "*************************************\n");
     while(i < MAX_HISTORY){
       memset(buffer[i], 0, MAX_BUFFER_LEN*unit_size);
       success_flag = history(buffer[i], i);
       if(success_flag == 0){
         if(i < 10)
           printf(1, "0%d    %s\n", i, buffer[i]);
         else
           printf(1, "%d    %s\n", i, buffer[i]);
       }
       i++;
       if(success_flag != 0) break;
     }
     printf(1, "*************************************\n");
     continue;
   }
```

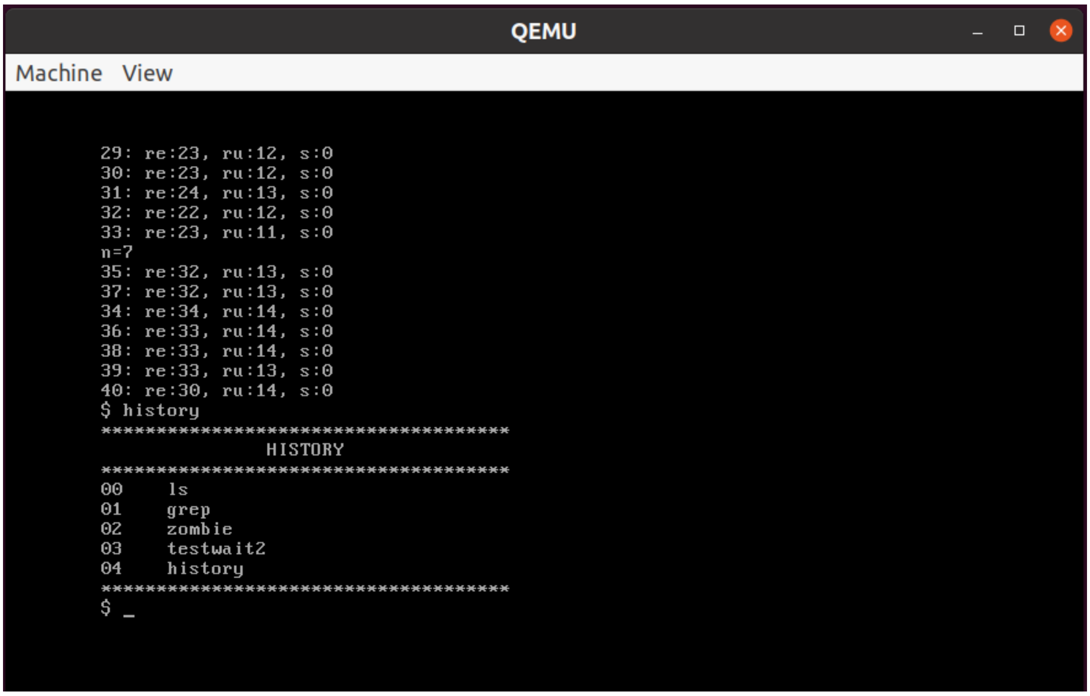
This utilises the sys_history function which is implemented in console.c. According to the assignment, it should have 2 inputs : a pointer to a buffer that will store the requested history and the index requested. The function should return 2 if history ID is illegal, 1 if no history exists, and 0 else.

Code :

```
int sys_history(void) {
   char *myBuffer;
   int id;

   // fetching parameters - namely Buffer and ID
   int fetchBufferIllegal = argstr(0, &myBuffer) < 0;
   int fetchIDIllegal = argint(1, &id) < 0;

   // checking whether arguments are fetched correctly
   if (fetchBufferIllegal || fetchIDIllegal) {
       return 1;
   }

   // checking whether the ID is positive and does not exceed pre
defined
   // history limit
```

```
    int illegalHistory = ((id < 0) || (id >= MAX_HISTORY));
    if (illegalHistory) {
        return 2;
    }

    // checking whether we have requested history or not
    int noHistory = id >= histCnt;
    if (noHistory) {
        return 1;
    }

    // copying from history buffer to local buffer using memmove
    int unit_size = sizeof(char);
    memmove(myBuffer, history_buffer[id], MAX_BUFFER_LEN * unit_size);
    return 0;
}
```

As the system call uses arguments given by the user, they cannot be directly passed as parameters and we need argstr and argint functions. If any of these are fetched incorrectly we exit by returning 1.
Then we check whether the given ID is legal, i.e. is positive and less than 16. Else we return 2.
Now if the given ID is greater than histCnt, the total number of history commands stored, we return 1.
Else we copy the contents of history_buffer at given index id to the buffer variable and return 0.
Output :
The image shows history along with their IDs, the larger the index - the more recent it is.



Task 2 : Statistics

Main observations:

- The allocproc() function is responsible for creating an entry for the process in the process table, apart from allocating space for the process and initialisation. Hence, the process time fields should be initialised in this function.

- The clock ticks generated by the hardware are served through interrupts to the cpu. Hence, the process statistics updating code must be called by this interrupt handler.

- Since both the ticks variable which counts the total number of ticks, and ptable, the process table are shared across cpu cores, they are protected by tickslock and ptable.lock respectively.

We first add the four fields, ctime, retime, rutime and stime to the proc struct in proc.h, which will store the relevant per-process statistics.

We initialise these values during process creation itself in the allocproc() function in proc.c. Here while the process itself is allotted an empty spot in process table and being initialised, it sets the time values as 0 :

```
p->stime = 0;
p->retime = 0;
p->rutime = 0;
```

We use the current value of ticks as creation time. The variable itself is protected by a lock due to it being shared across cores.

```
uint xticks;

acquire(&tickslock);
xticks = ticks;
release(&tickslock);

p->ctime = xticks;
```

We also create an updateProcTime function which is responsible for updating all the relevant times for each process. Note that it acquires the ptable.lock, the process table lock before updating.

```
void updateProcTime(void){
 struct proc *p;
 acquire(&ptable.lock);
 for(p = ptable.proc; p < &ptable.proc[NPROC]; ++p){
   if(p->state == UNUSED)
     continue;
   if(p->state == SLEEPING)
     p->stime++;
```

```
    else if(p->state == RUNNABLE)
      p->retime++;
    else if(p->state == RUNNING)
      p->rutime++;
  }
  release(&ptable.lock);
}
```

The following code is added in trap.c to handle updation of process times. Here, the condition cpuid() == 0 is important, as otherwise all the cores will increment the ticks variable for a single clock tick instead of it being incremented only once per clock tick. The increment itself is a critical section, hence protected by the tickslock.

```
case T_IRQ0 + IRQ_TIMER:
   if(cpuid() == 0){
     updateProcTime();
     acquire(&tickslock);
     ticks++;
     wakeup(&ticks);
     release(&tickslock);
   }
   lapiceoi();
   break;
```

Then changes are made to the following files and sys_wait2 system call is added

a) syscall.h
b) syscall.c
c) user.h
d) usys.S

```
int sys_wait2(void)
{
  int *retime;
  int *rutime;
  int *stime;

  if(argptr(0, (void*)&retime, 2*sizeof(retime[0])) < 0)
    return -1;
  if(argptr(1, (void*)&rutime, 2*sizeof(rutime[0])) < 0)
    return -1;
  if(argptr(2, (void*)&stime, 2*sizeof(stime[0])) < 0)
    return -1;

  return wait2(retime, rutime, stime);
}
```

wait2 is defined as follows in proc.c :

```
int wait2(int *retime, int *rutime, int *stime)
{
 struct proc *p;
 int havekids, pid;
 struct proc *curproc = myproc();
  acquire(&ptable.lock);
 for(;;){
   // Scan through table looking for exited children.
   havekids = 0;
   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
     if(p->parent != curproc)
       continue;
     havekids = 1;
     if(p->state == ZOMBIE){
       // Found one.
       pid = p->pid;

       //extension, gathering time stats
       *retime = p->retime;
       *rutime = p->rutime;
       *stime = p->stime;

       kfree(p->kstack);
       p->kstack = 0;
       freevm(p->pgdir);
       p->pid = 0;
       p->parent = 0;
       p->name[0] = 0;
       p->killed = 0;
       p->state = UNUSED;
       release(&ptable.lock);
       return pid;
     }
   }

   // No point waiting if we don't have any children.
   if(!havekids || curproc->killed){
     release(&ptable.lock);
     return -1;
   }

   // Wait for children to exit.  (See wakeup1 call in proc_exit.)
   sleep(curproc, &ptable.lock);  //DOC: wait-sleep
 }
}
```

Firstly the process table is locked and then we iterate through the table to find any process which has children who have exited. The code is identical to that in pre defined wait() function except we update the following times :

```
        //extension, gathering time stats
        *retime = p->retime;
        *rutime = p->rutime;
        *stime = p->stime;
```

The test case file needs to be added as a user program and hence in the Makefile, _testwait2 is added to UPROGS. Then we add test cases in the testwait2.c file. There are 3 test case types :

1) Zombie Time Test:
   This checks that the time spent by a process in the zombie state does not affect its process stats. We first fork a child which executes a dummy loop, and call wait immediately. The second time, we fork a child, but now the parent sleep for 100 ticks before calling wait. The times of the child are identical in both cases. (A deviation of +-1 is possible due to locking and timing mechanisms)

```
Zombie time test
4: re:0, ru:13, s:0
5: re:0, ru:13, s:0
```

2) Parent Child Test
   This test checks that when a parent P generates child process C and immediately calls wait(), then if C runs for x ticks, then the sleeping time of P when C exits should be x ticks as well.

```
Parent Child Test
7: re:0, ru:19, s:0
6: re:0, ru:0, s:19
```

3) Multiple Processes Test
   Here, the main loop in each iteration forks n children, n varying from 1 to 6. The children execute a dummy loop and then exit. We then print the time spent in the ready queue (retime) and the time spent running by the processes.

   First, we note that xv6 currently does a round robin scheduling with the time quantum of a single tick. The system here is a dual core system, hence for n=1 and n=2, we expect the processes to have 0 retime and some positive rutime.

   For n=3, The following pattern will emerge: P1 and P2 execute, P3 and P1 execute, P2 and P3 execute, P1 and P2 execute and so on. Since for every 2 ticks a process, say P1 runs, it waits 1 tick before being executed again.

   Hence, we expect the waiting time of the processes to be approximately half of their running time. A similar analysis shows that for n=4, the waiting time will be almost equal to the running time. The same applies to the other values of n as well.

```
Multiple processes Test
n=1
8: re:0, ru:20, s:0
n=2
9: re:0, ru:19, s:0
10: re:0, ru:20, s:0
n=3
11: re:10, ru:20, s:0
12: re:9, ru:21, s:0
13: re:10, ru:19, s:0
n=4
16: re:19, ru:18, s:0
14: re:19, ru:20, s:0
15: re:19, ru:19, s:0
17: re:19, ru:19, s:0
$
```