ASSIGNMENT 2(A)

Group 2 :-
1)Ritish Bansal 190101076
2)Suryansh Singh 190101089
3)Anant Shankhdhar 190101011
4)Mayank Chandak 190101052

Task 1

1.1 Caret Navigation:-

Files edited:-

1)**console.c** - created the following definitions

1. **KEY_LF** - Refers to the left arrow key
2. **KEY_RF** - Refers to the right arrow key

```
#define KEY_LF          0xE4
#define KEY_RF          0xE5
```

Created the following variable in the struct **input:-**

1. **uint pos :-** refers to the current position of the cursor in the buffer

```
struct {
  char buf[INPUT_BUF];
  uint r;  // Read index
  uint w;  // Write index
  uint e;  // Edit index
  uint pos; // Added by me current index
} input;
```

Implemented the following functions:-

1. **void move_back_cursor():-**
   a. This function is invoked when we wish to move the cursor backwards using the back arrow key.
   b. In the function we first get the current position of the cursor in the console in the pos variable. Then because we are moving back, we will decrement pos. At the end the cursor is reset to the new position pos we got from decrementing .
   c. The function implementation is given below:-

```
205
206 void move_back_cursor(){
207   int pos;
208
209   // get cursor position
210   outb(CRTPORT, 14);
211   pos = inb(CRTPORT+1) << 8;
212   outb(CRTPORT, 15);
213   pos |= inb(CRTPORT+1);
214
215   // move back
216   pos--;
217
218   // reset cursor
219   outb(CRTPORT, 15);
220   outb(CRTPORT+1, (unsigned char)(pos&0xFF));
221   outb(CRTPORT, 14);
222   outb(CRTPORT+1, (unsigned char )((pos>>8)&0xFF));
223   //crt[pos] = ' ' | 0x0700;
224
225 }
```

2. **void move_forward_cursor():-**
    a. This function is invoked when we wish to move the cursor forwards using the forward arrow key.
    b. In the function we first get the current position of the cursor in the console in a variable pos. Then because we are moving forward, we will increment the pos . At the end the cursor is reset to the new position pos we got after incrementing.
    c. The function implementation is given below:-

```
void move_forward_cursor(){
  int pos;

  // get cursor position
  outb(CRTPORT, 14);
  pos = inb(CRTPORT+1) << 8;
  outb(CRTPORT, 15);
  pos |= inb(CRTPORT+1);

  // move back
  pos++;

 // reset cursor
  outb(CRTPORT, 15);
  outb(CRTPORT+1, (unsigned char)(pos&0xFF));
  outb(CRTPORT, 14);
  outb(CRTPORT+1, (unsigned char )((pos>>8)&0xFF));
  //crt[pos] = ' ' | 0x0700;


}
```

3. **void insert_char(int c,int counter):-**
    a. This function is invoked when we enter a character.
    b. The function takes input the character **c** and an integer **counter** which is equal to negative of the distance of current cursor position and the end of the text.
    c. We store the current position of the cursor in the variable pos.
    d. First, we shift all characters from current position to end of text 1 position forward after we store character at pos.
    e. Finally pos is incremented by 1 as we added a new character. At the end the cursor is reset to the new position from incrementing pos.
    f. The implementation is giving below:-

```
void insert_char(int c,int counter){
  int pos;

  // get cursor position
  outb(CRTPORT, 14);
  pos = inb(CRTPORT+1) << 8;
  outb(CRTPORT, 15);
  pos |= inb(CRTPORT+1);

  for(int i=pos-counter; i>=pos; i--){
      crt[i+1]=crt[i];
  }
  crt[pos] = (c&0xff) | 0x0700;

  // move cursor to next position
  pos += 1;

  outb(CRTPORT, 14);
  outb(CRTPORT+1, pos>>8);
  outb(CRTPORT, 15);
  outb(CRTPORT+1, pos);
}
```

4. **void remove_char(int counter):-**
    a. This function is invoked when we enter the backspace key in the kernel.
    b. The function takes an integer **counter** which is equal to the negative of the distance of the current cursor position and the end of the text.
    c. We store the current position of the cursor in the variable pos.
    d. All the characters from current position to the text are shifted 1 position backwards x`removing the character to be deleted and replacing it with the next character .
    e. Finally the last character is changed to blank and pos is decremented as the new cursor will be 1 position behind
    f. Finally we reset the cursor to its new position
    g. The implementation is given below:-

```
void remove_char(int counter){
  int pos;

  // get cursor position
  outb(CRTPORT, 14);
  pos = inb(CRTPORT+1) << 8;
  outb(CRTPORT, 15);
  pos |= inb(CRTPORT+1);

  // move back
  for(int i=pos-1; i<=pos-counter-1; i++){
     crt[i]=crt[i+1];
  }
  crt[pos-counter]=' ' | 0x0700;
  // reset cursor
  pos--;
  outb(CRTPORT, 15);
  outb(CRTPORT+1, (unsigned char)(pos&0xFF));
  outb(CRTPORT, 14);
  outb(CRTPORT+1, (unsigned char )((pos>>8)&0xFF));
}
```

Edited the following functions:-

1. **void consoleintr(int (*getc)(void)):-**
    a. Added the case for remove_char when we are removing a character at a position

```
  case C('H'): case '\x7f':  // Backspace
   if(input.pos==input.e){
    if(input.e != input.w){
      input.e--;
      input.pos--;
      consputc(BACKSPACE);
    }
    }
    else if(input.pos!=input.w){
         remove_char(back_counter);
        for(int i=input.pos-1; i<=input.e-1; i++){
          input.buf[i]=input.buf[i+1];
        }
        input.e--;
        input.pos--;
    }
```

b. Added cases for moving the cursor forward and back within the text

```
case KEY_LF:
  if(input.pos>input.r){
      input.pos--;
      back_counter--;
      move_back_cursor();
  }
  break;
case KEY_RF:
  if(input.pos<input.e){
       input.pos++;
       back_counter++;
       move_forward_cursor();
  }
  break;
```

c. Added the condition for adding a new character at a given position inside

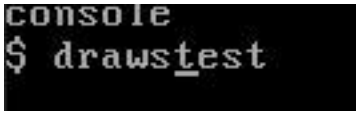
```
default:
  if(c != 0 && input.e-input.r < INPUT_BUF){
    c = (c == '\r') ? '\n' : c;
    if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
    input.buf[input.e++ % INPUT_BUF] = c;
    consputc(c);

    back_counter=0;

      input.w = input.e;
      input.pos=input.e;
      wakeup(&input.r);
    }
    else{
        if(back_counter==0){
          input.buf[input.e++ % INPUT_BUF] = c;
          input.pos++;
    consputc(c);
        }
        else{
          insert_char(c, back_counter);
          for(int k=input.e+1; k >= input.pos; k--){
          input.buf[(k + 1) % INPUT_BUF] = input.buf[k % INPUT_BUF];
          }

        //insert
        input.buf[(input.pos) % INPUT_BUF] = c;

        input.e++;
        input.pos++;

        //insert the char into CRT propoly position

        }
    }
  }
```

Output:-

Tests

1) We first write a dummy text in the console and use the arrow keys to move the cursor . We observe that the cursor moves as directed and upto the end of the text forwards and starting of the text backwards

2) Next we try adding a character at a middle position by navigating to that position and adding a character. We observe that the character gets added and the cursor moves 1 position forward as expected



3) Next we try deleting a character. We navigate to the position where we want to delete and press backspace. The character gets deleted and the cursor moves 1 step back.Nothing happens if we are at end of text for backspace.

1.2 Shell History Ring:-

Files edited:-

1)console.c:- created new definitions

1. MAX_HISTORY :- The maximum number of commands that can stay in the history. 16 in our case
2. MAX_COMMAND_LENGTH:- The maximum length of the command string i.e the size of the buffer. 128 in our case
3. KEY_UP :- Refers to the upward arrow key
4. KEY_DN :- Refers to the downward arrow key
5. The definitions are as follows:-

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

```
#define KEY_UP          0xE2
#define KEY_DN          0xE3
```

Created the following variables:-

1. **char command_history[MAX_HISTORY][MAX_COMMAND_LENGTH] :-** The history is stored as a 2d array . A maximum of 16 commands with length upto 128 can stay in history.
2. **int historycounter -** contains the number of items in history currently
3. **int currentcommandid -** contains the index of the current command in history
4. The variables declared are as follows:-

```
char command_history[MAX_HISTORY][MAX_COMMAND_LENGTH];
int historycounter=0;
int currentcommandid=0;
```

Implemented the following functions:-

1. **int history(char* buffer,int historyId):-**
    a. The function performs the function of copying the contents of history corresponding to the historyId to the buffer supplied from the system call.The parameters are the buffer as a char array and the historyId. This function is required for system call.

b. If the historyId is invalid(<0 or >=16) then the function returns 2. If the historyId is greater than the number of elements in history it returns 1 . If everything is fine then it copies the history into the buffer and returns 0.
c. The implementation is as follows:-

```
int history(char* buffer, int historyId){

   if(historyId<0 || historyId>=MAX_HISTORY){
         return 2;
   }
   if(historyId>=historycounter){
       return 1;
   }
   memmove(buffer,command_history[historyId],MAX_COMMAND_LENGTH*sizeof(char));
   return 0;
}
```

2. **void add_history(char* command):-**
   a. The function performs the function of adding the command character array into the history array.The parameter is only the command we wish to add in history at the end.
   b. The implementation is as follows:-

```
void add_history(char *command){
    if(command[0]!='\0'){
        int length=strlen(command);
        if(length>MAX_COMMAND_LENGTH){
           length=MAX_COMMAND_LENGTH-1;
        }
        if(historycounter<MAX_HISTORY){
           historycounter++;
        }
        else{
           for(int i=0; i<MAX_HISTORY-1; i++){
             memmove(command_history[i],command_history[i+1],sizeof(char)*MAX_COMMAND_LENGTH);
           }

        }
        memmove(command_history[historycounter-1],command,sizeof(char)*length);
        command_history[historycounter-1][length]='\0';
        currentcommandid=historycounter;
    }

}
```

Edited the following functions:-

1. **void consoleintr(int (*getc)(void)):-**
   a. Added the cases when we press the upward or downward arrow key. The implementation is as follows:-

```
case KEY_UP:
  if(currentcommandid==0){ }
  else if(currentcommandid>0){
      for(int i=input.pos; i<input.e; i++){
        move_forward_cursor();
      }
      while(input.e>input.w){
        input.e--;
        all_remove_char();
      }
      for(int i=0; i<strlen(command_history[currentcommandid-1]); i++){
         x=command_history[currentcommandid-1][i];
         cgaputc(x);
         input.buf[input.e++]=x;
      }
      currentcommandid--;
      input.pos=input.e;
  }
  break;
case KEY_DN:
  if(currentcommandid==historycounter-1){
      for(int i=input.pos; i<input.e; i++){
        move_forward_cursor();
      }
      while(input.e>input.w){
        input.e--;
        all_remove_char();
      }
      currentcommandid=historycounter;
  }
  else if(currentcommandid<historycounter-1){
      for(int i=input.pos; i<input.e; i++){
        move_forward_cursor();
      }
      while(input.e>input.w){
        input.e--;
        all_remove_char();
      }
      for(int i=0; i<strlen(command_history[currentcommandid+1]); i++){
         x=command_history[currentcommandid+1][i];
         cgaputc(x);
         input.buf[input.e++]=x;
      }
      currentcommandid++;
      input.pos=input.e;

  }
  break;
```

b. Inside the default case, we will add the **add_history** call whenever we enter a new command. We will perform add_history() after the enter key is pressed and the buffer is ready.

```
default:
  if(c != 0 && input.e-input.r < INPUT_BUF){
    c = (c == '\r') ? '\n' : c;
    if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
    input.buf[input.e++ % INPUT_BUF] = c;
    consputc(c);

    back_counter=0;
    for(int i=input.w,k=0; i<input.e-1; i++,k++){
         buffer[k]=input.buf[i % INPUT_BUF];
    }
    buffer[(input.e-1-input.w)%INPUT_BUF]='\0';

    add_history(buffer);
```

2)**Sysproc.c**:- Implemented the **int sys_history(void)** call

1. The function returns -1 if the if the buffer size is invalid or greater than max possible . Otherwise we call the history function for the historyId and buffer specified

```
int sys_history(void){
    char *buffer;
    int historyId;
    int size=128;
    if(argint(1,&historyId)<0){
       return -1;
    }
    if(argptr(0,(char **)&buffer,size)<0){
       return -1;
    }
    return history(buffer,historyId);

}
```

3)**Syscall.h** :- Added the history system call to the definitions

```
#define SYS_history 23
```

4)**Syscall.c** :- Added the SYS_history call defined in syscall.h to the syscall array and added sys_history call implemented in sysproc.c as external function

```
extern int sys_history(void);
```

```
[SYS_history] sys_history,
```

5)**usys.S**:- Added the SYSCALL(history) to be called

6)**user.h**:- Added the main system call **int history(char*, int)**

7)**defs.h**:- history defined in console is defined as it will be available to sys_proc.c

```
24  int             history(char*, int);
```

Running the history command - Files Created:-

1)**history.c**:-The file is for the user to use the history system call

1. We first declare the maximum number of commands in history (MAX_COMMANDS = 16) and the max size of the command (MAX_COMMAND_LENGTH = 128)
2. We create the buffer which is a 2D char array which will contain the history.
3. Then we run a loop MAX_COMMAND times to fill the loop using the history call for every index. If the value returned from the call is 0 it means everything went well and we add the history corresponding to that index to the buffer and print the buffer. Otherwise either index is invalid or index is greater than number of commands in history in that case we don't add it.
4. The code is shown below:-

```c
#include "types.h"
#include "stat.h"
#include "user.h"

#define MAX_HISTORY                (16)      /*the max number of the comand
    histories*/
#define MAX_COMMAND_LENGTH         (128)     /*the max length of the comand*/


int main(int argc, char *argv[])
{
    char buffer[MAX_COMMAND_LENGTH];
    int ret = 0;
    for(int i = 0; i < MAX_HISTORY && ret==0; i++){$
        memset(buffer, 0, 128 * sizeof(char));

        ret = history(buffer, i);

        if(ret == 0){
            printf(1, "%s\n", buffer);
        }
    }

    exit();
}
```

Files edited :-

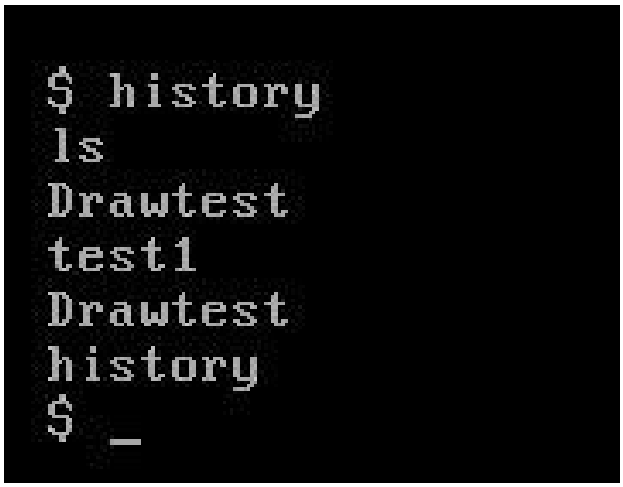
1)**Makefile**:- Added our testing file **history.c** to the list of user programs **UPROGS**

```
UPROGS=\
        _cat\
        _echo\
        _forktest\
        _grep\
        _init\
        _kill\
        _ln\
        _ls\
        _mkdir\
        _rm\
        _sh\
        _stressfs\
        _usertests\
        _wc\
        _zombie\
        _Drawtest\
        _history\
```

Test for the history command:-

1)For the simple test we run few commands. After this we run history command. We see the commands we ran above are visible in history.

The output of the history command is as follows:-



2)Next we first run 17 commands. After that we press upward arrow key 17 times. We see we can go to prev command only 16 times after that nothing happens, This is because only 16 commands can be stored in history. For moving the downward key we can move upto clear text so that we can write new command and execute it.

Task 2:- Statistics

Files edited:-

1)proc.h:-

1. Inside the struct **proc** we have added the ctime,stime,retime and runtime variables, the updated struct looks like this.

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)

  int ctime;
  int stime;
  int retime;
  int rutime;
};
```

2. We define the prototype of **void Update_ticks()** and **int wait2(int *retime,int *rutime,int *stime)** functions we implemented in **proc.c**

```
void Update_Ticks();
int wait2(int *retime, int *rutime, int *stime);
```

2)proc.c :-

1. Inside the **static struct proc* allocproc(void) function** we initialised the ctime, stime, retime, rutime variables for a process p in the ptable.

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->ctime = ticks;
  p->stime = 0;
  p->retime = 0;
  p->rutime = 0;

  release(&ptable.lock);
```

2. We implement the function **void Update_Ticks()** that will be executed whenever the ticks increment. This function is used to update the retime,stime and rutime whenever to tick increments. If the process is SLEEPING, we increment the stime, if RUNNABLE, we increment the retime and if it is RUNNING, we increment the rutime.

```
void Update_Ticks()
{
    struct proc *p;
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        switch(p->state)
        {
            case SLEEPING:
            p->stime++;
            break;

            case RUNNABLE:
            p->retime++;
            break;

            case RUNNING:
            p->rutime++;
            break;

            default:
            ;
        }
    }
    release(&ptable.lock);
}
```

3. We implement the wait2 system call function **int wait2(int *retime,int *rutime,int *stime)** which extend the **wai**t function and has the extra functionality to store the process variables retime,stime and rutime so that they can be displayed when the user program is

executed.The statements for storing these variables are shown in line 333-335 below

```
318 int wait2(int *retime, int *rutime, int *stime) {
319   struct proc *p;
320   int havekids, pid;
321   struct proc *curproc = myproc();
322
323   acquire(&ptable.lock);
324   for(;;){
325     // Scan through table looking for zombie children.
326     havekids = 0;
327     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
328       if(p->parent != curproc)
329         continue;
330       havekids = 1;
331       if(p->state == ZOMBIE){
332         // Found one.
333         *retime = p->retime;
334         *rutime = p->rutime;
335         *stime = p->stime;
336         pid = p->pid;
337         kfree(p->kstack);
338         p->kstack = 0;
```

3)traps.c:- Added the call to Update_ticks() whenever the ticks are incremented

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

4)sysproc.c:- Add the **int sys_wait2(void)** function

1. The function first checks if the the retime, rutime and stime are valid (i.e >= 0).If not so then it returns -1.
2. Otherwise we make a call to the wait2() function in the proc.c file with the variable *retime,*rutime and *stime declared in the function . The implementation is shown below

```
int sys_wait2(void)
{
  int *retime, *rutime, *stime;
  if (argptr(0, (void*)&retime, sizeof(retime)) < 0)
    return -1;
  if (argptr(1, (void*)&rutime, sizeof(rutime)) < 0)
    return -1;
  if (argptr(2, (void*)&stime, sizeof(stime)) < 0)
    return -1;
  return wait2(retime, rutime, stime);
}
```

5)**syscall.h**:- Added the definition for the SYS_wait2 call

```
#define SYS_wait2  24
```

6)**syscall.c:-** Declared the **int sys_wait2(void)** implemented in sysproc.c as extern function and added the **SYS_wait2** call to the system calls array.

```
extern int sys_wait2(void);
```

```
[SYS_wait2]   sys_wait2,
```

7)**user.h**:- Added the definition for our main system call **int wait2(int *retime,int *rutime,int *stime)**

8)**usys.S**:- Added the **SYSCALL(wait2)** statement

Testing the implementation:-

Files created :-

1)**test1.c:-**

The file contains or testcase for the statistics

1. We create two child processes using the fork() system call. The first child process sleeps for 100 ticks and then ends. The second child process first sleeps for 200 ticks and then runs a loop before terminating.
2. We expect that first we see the statistics for the first process for which the stime should be 100 and retime and rutime should be 0 as the process was sleeping the entire time
3. For the second process we expect that first we should see the output of the process which is integers in range 0 to 9999. After that in statistics. the stime =200 as the process was sleeping for 200 ticks and retime = 0 and a non zero value for rutime.
4. After this we use our wait2 system call and printf command to show the statistics for these processes.

```c
#include "types.h"
#include "user.h"
#include "spinlock.h"



int
main(int argc, char *argv[])
{
  if (fork() == 0) {
    sleep(100);
    exit();
  }



  if(fork() == 0) {

    sleep(200);
    for(int i=0;i<10000;i++)
    {
    printf(1, "%d ", i);
    }
    printf(1, "\n");

    exit();

  }

  int retime, rutime, stime;

  printf(1, "%d\n",wait2(&retime, &rutime, &stime));
  printf(1, "retime=%d rutime=%d stime=%d\n", retime, rutime, stime);
  printf(1, "turnaround time=%d\n", retime+rutime+stime);

  printf(1, "%d\n", wait2(&retime, &rutime, &stime));
  printf(1, "retime=%d rutime=%d stime=%d\n", retime, rutime, stime);
  printf(1, "turnaround time=%d\n", retime+rutime+stime);
  exit();

}
```

Files edited:-

1)**Makefile:-** Added the test1 to user programs UPROGS

```
UPROGS=\
        _cat\
        _echo\
        _forktest\
        _grep\
        _init\
        _kill\
        _ln\
        _ls\
        _mkdir\
        _rm\
        _sh\
        _stressfs\
        _usertests\
        _wc\
        _zombie\
        _Drawtest\
        _history\
        _test1\
```

Running the tests:-

1) On performing ls we can see test1 is visible as a user program . history is also visible

```
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16340
echo           2 4 15192
forktest       2 5 9508
grep           2 6 18560
init           2 7 15780
kill           2 8 15220
ln             2 9 15080
ls             2 10 17708
mkdir          2 11 15324
rm             2 12 15300
sh             2 13 27936
stressfs       2 14 16212
usertests      2 15 67320
wc             2 16 17076
zombie         2 17 14892
Drawtest       2 18 15288
history        2 19 15508
test1          2 20 15852
console        3 21 0
$
```

2) We run the test1 command and the output is as follows:-

```
$ test1
5
retime=0 rutime=0 stime=100
turnaround time=100
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
6
retime=0 rutime=684 stime=200
turnaround time=884
$
```

We can observe that for the first process stime = 100 and retime and rutime = 0 . This is as expected because the process did not run or was runnable and was in sleep for 100 ticks before terminating .
After the first process we observe the second process occurred and we see the 10000 integers got printed which is as expected .
Finally after the process terminates we can see the statistics for the second process. As we gave 200 ticks for sleep we can see that clearly from the output. Other than that retime = 0 and rutime = 684. Which shows the process was running the entire time/
The turnaround time is the sum of all three values as expected/
We can see statistics for both the processes

Appying Patch File:-

The command to apply main final patch file is :- **git apply --reject --whitespace=fix patch.txt**
The command to apply a patch file for specific task is :- **patch -ruN -strip -d xv6-public < [filename].patch**
Note :- -strip is required to prevent git logging issues