CS 344: OS Lab

Assignment - 2A

==

Group-9:

1. Sai Ashrritth Patnana 190101061
2. Bharath Bhukya 190101024
3. Adduri Sai Sri Datta 190101003
4. Amancha Jagruth 190101010

Task-1:

Before implementing caret function, we need to make some changes to the functions which manage painting characters on screen.

When we enter a character, it is passed to **consoleintr()** function as an argument, an to print them on the screen this function calls **constputc()** and passes the character onto it. This constputc checks if the system is in a panicked state or not, and then it prints the character or shifts the caret's position on our linux terminal using uartputc(). And at the end it calls **cgaputc()** and passes the character.

```c
void
consputc(int c)
{
  if(panicked){
    cli();
    for(;;)
      ;
  }

  switch (c) {
    case BACKSPACE:
      uartputc('\b'); uartputc(' '); uartputc('\b');  // uart is writing to the linux shell
      break;
    case LEFT_ARROW:
      uartputc('\b');
      break;
    default:
      uartputc(c);
  }
  cgaputc(c);
  // uartputc prints to Linux's terminal and cgaputc prints to QEMU's terminal
}
```

```c
static ushort *crt = (ushort*)P2V(0xb8000);  // CGA memory

static void
cgaputc(int c)
{
  int pos;

  // Cursor position: col + 80*row.
  outb(CRTPORT, 14);
  pos = inb(CRTPORT+1) << 8;
  outb(CRTPORT, 15);
  pos |= inb(CRTPORT+1);

  switch(c) {
    case '\n':
      pos += 80 - pos%80;
      break;
    case BACKSPACE:
      if(pos > 0) --pos;
      break;
    case LEFT_ARROW:
      if(pos > 0) --pos;
      break;
    default:
      crt[pos++] = (c&0xff) | 0x0700;  // black on white
  }

  if(pos < 0 || pos > 25*80)
    panic("pos under/overflow");

  if((pos/80) >= 24){  // Scroll up.
    memmove(crt, crt+80, sizeof(crt[0])*23*80);
    pos -= 80;
    memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
  }

  outb(CRTPORT, 14);
  outb(CRTPORT+1, pos>>8);
  outb(CRTPORT, 15);
  outb(CRTPORT+1, pos);
  if (c == BACKSPACE)
    crt[pos] = ' ' | 0x0700;
}
```

Now this function makes the required changes in the CGA memory which is the memory used to store the character on screen.

Now this **CGA memory** is a linear memory of size 25*80 (rows * columns). Now in the beginning of the cgaputc function, we retrieve positon at the editing occurs from a **CRTPORT** whose address is 0x3d4 at pin 14 and pin 15 in 16-bits. We retrieve that value through CRTPORT+1 and insert our character at that position. So if the character is BACKSPACE or LEFTARROW, we will just decrease position by 1, otherwise we insert our character at position and increase it by 1. If it is a backspace we are doing then we insert ' '. Now if the position is going into the last line then we will shift the whole CGA memory arrey back by 80 units so that in the screen it looks like the whole screen moved upwards, and we will set all values after to null. And at the end we will update the value in CRTPORT in pin 14,15. So now this constputc will act as our basic function to manipulate the output on the screen safely and we will use this as suited for our use.

Now we implemented a struct input, which stores a copy of the character on screen. It is used to copy characters which are on screen instead of taking directly from screen memory. So every time we make a change to screen memory we make a change here also.

1.1: Caret Navigation:

The carter navigation should satisfy the following functionalities.

1.Carter movement to Left and Right:

To move the arrow left and right it is enough to change the code in console.c such that on entering the arrow keys in console the carter changes its position. We change the value of input.e parameter to make movement on the console command. If the character input is left arrow the value of input.e decreases.

```
        case LEFT_ARROW:
          if (input.e != input.r) {
            input.e--;
            consputc(c);
          }
          break;
```

```
        case RIGHT_ARROW:
          if (input.e < input.rightmost) {
            consputc(input.buf[input.e % INPUT_BUF]);
            input.e++;
          }
          else if (input.e == input.rightmost){ // This line add the cursor at the end ogf the line.
            consputc(' ');
            consputc(LEFT_ARROW);
          }
          break;
```

Console changes have been listed below –

Original pic

```
Drawtest        2 18 15964
console         3 19 0
$ hello world
```

if left arrow pressed 6 times

```
Drawtest        2 18 15964
console         3 19 0
$ hello world
```

if right arrow for 3 times

```
Drawtest        2 18 15964
console         3 19 0
$ hello world
```

2.Carter movement to next line:

If the input character is "\n" the it has no other option but to move to next line. The carter moves below upon entering Enter Key.

```
if(c == '\n' || c == C('D') || input.rightmost == input.r + INPUT_BUF){
  saveCMDinHistoryMem(); // when enter is entered we saving that command to historyMem
  input.w = input.rightmost;
  wakeup(&input.r);
}
```

Console changes have been listed below-

```
$
$
$ we are about to press enter
```

```
$ we are about to press enter
exec: fail
exec we failed
$
```

3.Buffer Editing:

-> On entering "BACKSPACE":

The character on the left of the carter are deleted and to which are on the right of the carter are just shifted to the left.

Upon entering the "BACKSPACE", the function "leftshiftBuffer" is called that shifts all input characters after the carter position to the left and backspaces to left as number of times it is pressed.

```
case C('H'): case '\x7f':  // Backspace
  if (input.rightmost != input.e && input.e != input.r) { // caret isn't at the end of the line
    shiftbufleft(); // shifting buffer to one position left.
    break;
  }
  if(input.e != input.r){ // caret is at the end of the line - deleting last char
    input.e--;
    input.rightmost--;
    consputc(BACKSPACE);
  }
  break;
```

```
void shiftbufleft() {
  uint n = input.rightmost - input.e;
  uint i;
  consputc(LEFT_ARROW);
  input.e--;
  for (i = 0; i < n; i++) {
    char c = input.buf[(input.e + i + 1) % INPUT_BUF];
    input.buf[(input.e + i) % INPUT_BUF] = c;
    consputc(c); // repainting the screen.
  }
  input.rightmost--;
  consputc(' '); // delete the last char in line
  for (i = 0; i <= n; i++) {
    consputc(LEFT_ARROW); // shift the caret back to the left
  }
}
```

-> On entering Data:

The entry might consists of letters/digits/symbols. The entered data is placed at the position where the carter is currently in. The data is placed at the carter position and the carter is shifted to right.

```
        default:
          if(c != 0 && input.e-input.r < INPUT_BUF){
            c = (c == '\r') ? '\n' : c;
            if (input.rightmost > input.e) { // caret isn't at the end of
the line
              copybuffToBeShifted();
              input.buf[input.e++ % INPUT_BUF] = c;
              input.rightmost++;
              consputc(c);
              shiftbufright();
            }
            else {
              input.buf[input.e++ % INPUT_BUF] = c;
              input.rightmost = input.e - input.rightmost == 1 ?
input.e : input.rightmost;
              consputc(c);
            }
```

```
void shiftbufright() {
  uint n = input.rightmost - input.e;
  int i;
  for (i = 0; i < n; i++) {

    char c = buffToBeShifted[i];
    input.buf[(input.e + i) % INPUT_BUF] = c;
    consputc(c); // repaitning the screen.
  }
  // reset buffToBeShifted for future use
  memset(buffToBeShifted, '\0', INPUT_BUF);
  // return the caret to its correct position
  for (i = 0; i < n; i++) {
    consputc(LEFT_ARROW);
  }
}
```

Upon entering the data, the function “rightshiftBuffer” is called which shifts shift all the input characters present after the position where new data is entered to the right consequently shifting in the position of carter to the right.

Console changes have been listed below-

Some of editions of buffer is provided below

1. Considering arbitrary position of carter (here it is ‘a’)

```
$ initial state
```

2. Entering data from the arbitrary position (here it is [NEW ENTRY])

   The data is right shifted and new data entry is written

```
$ inti[NEW ENTRY]al entry
```

3. Deletion of Data (here the deleted data is ‘iti’)

   After removing characters the carter is left shifted with the data to the right of it

```
$ inal entry
```

1.2: Shell history ring:

First we are storing the previous 16 commands in the structure below.

```
// this struct stores the commands and its details.
struct {
  char CommandMemArr[MAX_HISTORY][INPUT_BUF];   // holds the actual command strings.
  uint lengthsArr[MAX_HISTORY];                 // this will hold the length of each command string.
  uint FinalCMdIndex;                          // the index of the last command entered to history.
  int TotalCMDsInMem;                           // total number of commands executed from the system boot.
  int currentPosition;                          // no. of skips in history array while toggling up and down arrow.
} HistoryMem;
```

As shown in the comments, the following parameter have its following uses. Now when a user hits enter in the console, we will call a function named **saveCMDinHistoryMem()** which is implemented in consoleread(). This function saves the command in its respective index, and it stores them in a cyclic manner so that if the total commands are more than 16 we will still get the previous 16 commands.

```
// This method saves the current command into the historyMem
void
saveCMDinHistoryMem(){
  HistoryMem.TotalCMDsInMem++; // counting the total no.of commands executed till now.
  uint l = input.rightmost-input.r -1;
  HistoryMem.FinalCMdIndex = (HistoryMem.FinalCMdIndex - 1) % MAX_HISTORY; // this step stores the commands in a cyclic manner if the memory is full.
  HistoryMem.lengthsArr[HistoryMem.FinalCMdIndex] = l;
  uint i;
  for (i = 0; i < l; i++) { //do not want to save in memory the last char '/n'
    HistoryMem.CommandMemArr[HistoryMem.FinalCMdIndex][i] =  input.buf[(input.r+i)%INPUT_BUF];
  }
  return;
}
```

First we have initialized the values of FinalCmdIndex, TotalCmdsInMem to 0 and currentPosition to -1 in consoleinit(). Now when saveCMDinHistoryMem() is called for the first time, it will set the FinalCMDIndex to 15 and store the command there. Now when it is called again and again it will keep storing the commands at 14, 13, 12 … till 0. Now when we reach 0, i.e the memory is full, it will set the FinalCMDIndex at 15, and it overwrites the first command with the new 17th command. So in this way it stores its data bye cyclic through all the indexes. Fetching its data correctly in an orderly fashion is described in below sections. The saveCMDinHistoryMem also sets the currentPosition to -1, and increments the TotalCMDsInMem by 1.

Now we have stored the previous commands, lets see how to access them by up/ down arrow and by a shell user program.

UP/DOWN ARROW:

So when a user hits the up arrow, we need to show the previous command which was executed.

So when a user first time hits an up arrow, we will do the following steps:

- First we will store the character which he typed before hitting up arrow in buffer named as oldBuf and its length in lengthOldBuf (line 215).
- Then we will increase currentPosition by 1, which makes it zero and then we will show the command which was stored at FinalCMDIndex - currentPosition, by first erasing the content on the screen and repainting the screen with the new characters.
- Now if the up arrow is hit again, we will continue doing step 2, and we will continue doing this every time the up arrow is hit until either currentPosition is equal to total no. of commands or it is equals to 16.

```
     case UP_ARROW:
      if (HistoryMem.currentPosition < HistoryMem.TotalCMDsInMem-1 && HistoryMem.currentPosition < MAX_HISTORY-1 ){
         // current history means the oldest possible will be MAX_HISTORY-1
         earaseCurrentLineOnScreen(); // eraseing the whole line
         earaseContentOnInputBuf();   // erasing in input.buf
         if (HistoryMem.currentPosition == -1) // if it is the first toggle we make then the our written command  should be stored.
             copybuffToBeShiftedToOldBuf();
         HistoryMem.currentPosition++; // toggling by increasing out current position.
         tempIndex = (HistoryMem.FinalCMdIndex + HistoryMem.currentPosition) %MAX_HISTORY; // gives us the index of currentposition'th index from the recent command.
         copyBufferToScreen(HistoryMem.CommandMemArr[ tempIndex]  , HistoryMem.lengthsArr[tempIndex]);
         copyBufferToInputBuf(HistoryMem.CommandMemArr[ tempIndex]  , HistoryMem.lengthsArr[tempIndex]);
       }
       break;
```

So this shows the implementation of the code of the up arrow.

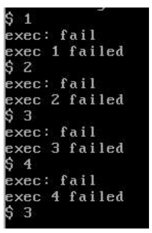
Now, when a user hits the down arrow, we will do the opposite, we will decrease the currentPosition value by 1 show command at the respective position and if it reaches 0 it means the next command we have to do will be the command the user wrote before accessing the Mem, i.e the memory stored in oldBuf. The code is shown below.

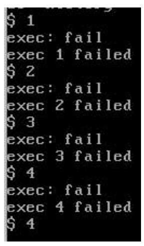
```
     case DOWN_ARROW:
       switch(HistoryMem.currentPosition){
         case -1:
           //does nothing
           break;
         case 0: // prints the string from oldbuff
           earaseCurrentLineOnScreen();
           copyBufferToInputBuf(oldBuf, lengthOfOldBuf);
           copyBufferToScreen(oldBuf, lengthOfOldBuf);
           HistoryMem.currentPosition--; // decreasing out current position.
           break;
         default:
           earaseCurrentLineOnScreen();
           HistoryMem.currentPosition--; // decreasing out current position.
           tempIndex = (HistoryMem.FinalCMdIndex + HistoryMem.currentPosition) % MAX_HISTORY;
           copyBufferToScreen(HistoryMem.CommandMemArr[ tempIndex]  , HistoryMem.lengthsArr[tempIndex]);
           copyBufferToInputBuf(HistoryMem.CommandMemArr[ tempIndex]  , HistoryMem.lengthsArr[tempIndex]);
           break;
       }
       break;
```

Lets see if it works correctly.

I will run commands 1,2,3,4 for simplicity and then hit up arrow twice and lets see what happens.(left image)

Now, lets hit down arrow once, (right image)





System call:

Now we will add a system call to xv6 named "history". So we will go through all the steps we need to do in to create a system call in syscall.c, syscall.h, user.h, usys.S and in sysproc.c we wrote the function void sys_history() like this

```c
int sys_history(void) {
  char *buffer;
  int historyId;

  if(argptr(0, &buffer, 1)==-1) return -1;

  if(argint(1, &historyId)==-1) return -1;

  return history(buffer, historyId);
}
```

Now this system call returns the value returned by the history(buffer, historyId) which is defined defs.h and implemented in console.c. Now this function copies the oldest historyId'th command into the buffer. It returns the values 0,1,2 for its respective cases as told in the question.

The history was implemented like this.

```c
int history(char *buffer, int historyId) {
  // this function returns command which was executed at historID+1 position in the stored MAX_HISTORY commands.
  if (historyId < 0 || historyId > MAX_HISTORY - 1)
    return -2;
  if (historyId >= HistoryMem.TotalCMDsInMem )
    return -1;
  memset(buffer, '\0', INPUT_BUF);
  uint temp;
  if(HistoryMem.TotalCMDsInMem > MAX_HISTORY){
    temp = HistoryMem.FinalCmdIndex - 1;
  }
  else{
    temp = MAX_HISTORY - 1;
  }
  temp = (temp - historyId) % MAX_HISTORY;
  memmove(buffer, HistoryMem.CommandMemArr[temp], HistoryMem.lengthsArr[temp]);
  return 0;
}
```

Now to implement a shell user program, we wrote the following code in int main() of sh.c and implemented a void printHistory() function which call the system call history and prints the command into the console.

```c
void printHistory(){
  uint i ;
  uint count = 0;
  for(i= 0; i <MAX_HISTORY; i++){
    if(history(getHistoryCommand, i) == 0){
      count++;
      printf(1, "%d: %s\n", count, getHistoryCommand);
    }
  }
  return;
}
```

```c
    if(buf[0] == 'h' && buf[1] == 'i' && buf[2] == 's'  && buf[3] == 't' && buf[4] == 'o' && buf[5] == 'r' && buf[6] == 'y' && buf[7] == '\n'){
      printHistory();
      continue;
    }
```

Now, output to check if the history is working or not.

First let's check by entering just 3 commands, for simplicity i am executing 1,2,3 as commands. (left image)

Now I will execute the same way upto 18 and see if the cycling is working and history function in console.c is extracting the data properly. (right image)

We can see that cycling is working.

```
init: starting sh
$ 1
exec: fail
exec 1 failed
$ 2
exec: fail
exec 2 failed
$ 3
exec: fail
exec 3 failed
$ history
1: 1
2: 2
3: 3
4: history
$
```

```
exec: fail
exec 17 failed
$ 18
exec: fail
exec 18 failed
$ history
1: 4
2: 5
3: 6
4: 7
5: 8
6: 9
7: 10
8: 11
9: 12
10: 13
11: 14
12: 15
13: 16
14: 17
15: 18
16: history
$
```

Task-2:

Functionality: To add system call wait2

Changes made in the files are as follows:

1. **syscall.h** :- System call SYS_wait2 is defined
2. **syscall.c** :- Created pointer sys_wait2 which points to the above defined system call vector in syscall.h and also declared external function sys_wait2(void) for the above system call.
3. **sysproc.c** :- Implemented the function SYS_wait2 in this file
4. **usys.s** :- Added user level system call for our function
   1. Used to connect user call to system call
5. **defs.h** :- Function wait2(int*,int*,int*) is defined, which is called when SYSCALL(wait2) is called by console
6. **proc.c** :- Implemented wait2 function
7. **proc.h** :- Extended proc struct with ctime,retime, rutime, stime fields

```
// Per-process state
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)
  uint ctime;                  // Process creation time
  uint stime;                  // process sleeping time
  uint retime;                 // process ready time
  uint rutime;                 // process running time
};
```

BLANK

Wait2 function is Implemented in proc.c

```c
int wait2(int *retime, int *rutime, int *stime) {
  struct proc *p;
  struct proc *curproc=myproc();

  int havekids, pid;
  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for zombie 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.
        // retrieving ready, run, sleep time
        *retime = p->retime;
        *rutime = p->rutime;
        *stime = p->stime;
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->state = UNUSED;
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->ctime = 0;
        p->retime = 0;
        p->rutime = 0;
        p->stime = 0;
        release(&ptable.lock);
        return pid;
      }
    }

    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
  }
}

```

For each clock cycle

updatestatistics function updates the parameters stime,retime,rutime, it is called in **trap.c** where the clock tick occurs and updates the ticks count.

```c
// this function runs when a tick occurs and is called in trap.c
void updatestatistics() {
  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);
}
```

User programs to make wait2 system call

1. Created Statistics.c in xv6-public, which contains main function to make system call and this function name is added in makefile in user defined programs
   _Statistics/ in UPROGS
   Statistics.c/ in EXTRA

Output:-

```
$ statistics
pid: 6, retime: 0, rutime: 0, stime: 0
pid: 5, retime: 0, rutime: 3, stime: 0
pid: 7, retime: 0, rutime: 0, stime: 0
pid: 4, retime: 0, rutime: 5, stime: 3
pid: 9, retime: 0, rutime: 0, stime: 0
pid: 8, retime: 0, rutime: 3, stime: 0
pid: 10, retime: 0, rutime: 0, stime: 0
$ _
```