भारतीय प्रौद्योगिकी संस्थान गुवाहाटी
INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI

CS344 - Assignment 2A - Group 11

TASK 1

Task 1.1 - Caret Navigation:

In this task we have to improvise the console such that when we pressed left arrow and right arrow there ASCII values should not be typed and instead of that it should move left and right accordingly. For this we have modified the file console.c in which we have added leftshift(), rightshift() and edited functions are cgaputc(), consputc() and consoleintr().

input struct (stucture for input buffer), has buffer memory, read index(r), write index(w), edit index(e). Without left and right arrows enabled edit index is always the last index of the current input entered by the user. Now when left arrow can be used edit index might not be the last index and we need to store this value for some usage, hence last index (l) has been added to input buffer stuct.

```
#define INPUT_BUF 128
struct {
  char buf[INPUT_BUF];
  uint r;  // Read index
  uint w;  // Write index
  uint e;  // Edit index
  uint l;  // Last letter(of current command) index
} input;
```

Figure 1: input buffer

The following functions have been modified:

- **consputc():** consputc is the function used to add a charecter at the current cursor in both the terminals (Qemu and Linux). In Consputc() there are two functions which are cgaputc() which is used to update the terminal display in qemu terminal where as uartputc() will update the Linux terminal.Here to update the Linux terminal When a LEFT arrow is clicked we have written uartputc(\b) which will move the cursor backwards and in case of RIGHT arrow, consputc(c) where c is the current character will do the job.

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

  if(c == BACKSPACE){
    uartputc('\b'); uartputc(' '); uartputc('\b');
  }else if(c == LEFT){
    uartputc('\b');
  }else{
    uartputc(c);
  }
  cgaputc(c);
}
```

Figure 2: consputc() - LEFT arrow on LINUX terminal updated at uartputc()

- **cgaputc():** Here we have added the code that when a LEFT arrow is clicked we have to reduce the pos by one place which will move the cursor in Qemu's terminal one unit left. But we put space in crt table only if the current character is BACKSPACE. Previously only editing happens at end of the line and putting space at end works all the time, but now since left arrow functionality is added there is no need to add a space in between the line, instead only the value of pos have ot be decreased by 1.

```
if(c == '\n')
  pos += 80 - pos%80;
else if(c == BACKSPACE){
  if(pos > 0) --pos;
}else if(c == LEFT){
  if(pos > 0) --pos;
}else{
  crt[pos++] = (c&0xff) | 0x0700;  // black on white
}
```

Figure 3: cgaputc() - when LEFT arrow is entered

```
171   outb(CRTPORT+1, pos);
172   if(c == BACKSPACE){
173     crt[pos] = ' ' | 0x0700;
174   }
```

Figure 4: ccgaputc() - to put space only for BACKSPACE

- **leftshift():** Previously backspace can be used only at the end of the line. Now, since left arrow can be used, it need not be only at the end. Hence when BACKSPACE is clicked somewhere in middle of the line the input buffer and the text on the screen has to be shifted a unit left to maintain consistency. This operation is facilitated using leftshift() function.
  Both left and right shifts preform similar operations. for loops in the function copies shifts the characters in the buffer leftwards. finally cursor is again taken back to the middle of the line(position depends) to give a better user exp.

```
438  // echo hello world
439  void leftShift(void){
440    int n = input.l - input.e;
441    int i = 0;
442
443    for(i=0;i<n;i++){
444      char tmp = input.buf[(input.e+1+i)%INPUT_BUF];
445      input.buf[(input.e+i)%INPUT_BUF] = tmp;
446      consputc(tmp);
447    }
448    consputc(' ');
449    for(i=0;i<=n;i++){
450      consputc(LEFT);
451    }
452  }
```

Figure 5: leftshift()

```
void rightShift(void){
  int n = input.l - input.e;
  int i;
  input.e++, input.l++;
  for(i=input.l-1;i>input.e;i--){
    input.buf[i%INPUT_BUF] = input.buf[(i-1)%INPUT_BUF];
  }
  input.buf[i%INPUT_BUF] = tmpchar;
  tmpchar = '\0';

  for(i = input.e;i<input.l;i++){
    consputc(input.buf[i%INPUT_BUF]);
  }
  for(i=0;i<n;i++){
    consputc(LEFT);
  }
}
```

Figure 6: rightshift()

- **rightshift():** We have implemented the rightshift() function such that at the middle of the cursor if we have typed something we have to move the complete buffer from that point to the rightmost index by one unit.So we ran a for loop to move right and update that character at that position and again move back the cursor to the appropriate position.

- **consoleintr():** It is the heart of the Task-1.1 this is place where when an interrupt in keyboard occurs this function would have been called and here for this task the following have been modifed:

  - **Case LEFT:** When left arrow is clicked if the edit index is not at the start we had decreased the edit index and we have called consputc(LEFT).if the edit index is at read index then if we have clicked left arrow , no need to do anything.
  - **Case RIGHT:** When right arrow is clicked if the edit index is not at the end we have put the same character again at that place and this will make the cursor to move one step that is the desired action of the right click and we had increased the edit index at last.If initially we are at rightmost point, the nothing to do.
  - **Case BACKSPACE:** If Backspace has been pressed at starting index we don't need to do anything and if it has been pressed in between then we will reduce both the edit index (input.e) and last index (input.l) by one unit and we will call the leftshift() function so that all the characters

```
case LEFT:
  if(input.e != input.w){
    input.e--;
    consputc(LEFT);
  }
  break;
case RIGHT:
  if(input.e != input.l){
    consputc(input.buf[(input.e)%INPUT_BUF]);
    // consputc(RIGHT);
    input.e++;
  }
  break;
```

Figure 7: LEFT and RIGHT inputs

```
case C('H'): case '\x7f':  // Backspace
  if(input.l != input.e && input.e != input.w){
    consputc(LEFT);
    input.e--;
    input.l--;
    leftShift();
  }else if(input.e != input.w){
    input.e--;
    input.l--;
    consputc(BACKSPACE);
  }
  break;
```

Figure 8: input is BACKSPACE

after edit index would shift left. If the position already at last then nothing to change but use a BACKSPACE directly.

```
default:
  if(c=='\n' || c=='\r'){
    input.e = input.l;
    saveToHistory();
  }
  if(c != 0 && input.e-input.r < INPUT_BUF){
    c = (c == '\r') ? '\n' : c;
    if(input.l > input.e){
      // if the inserted caracter is in the middle
      // the next to it string has to be moved a bit
      tmpchar = input.buf[input.e%INPUT_BUF];
      input.buf[input.e%INPUT_BUF] = c;
      consputc(c);
      rightShift();
    }else{
      input.buf[input.e% INPUT_BUF] = c;
      input.e++;
      input.l++;
      consputc(c);
    }
    if(c == '\n' || c == C('D') || input.l == input.r+INPUT_BUF){
      input.w = input.l;

      wakeup(&input.r);
    }
  }
  break;
```

Figure 9: Caption

– **Case DEFAULT:** Main change occurs only if a normal letter is entered at the middle of a line. If it is at the last the case was already written, now when chracter is to enter in middle it has to move the letters in the buffer by one unit right along with updating the console input accordingly. This is where rightshift() comes into picture and does the job. Also not that input.e and input.l have been separated and the functions that read and implement the input, they will normally read upto input.e, but now they have to read until input.l (as it the lst character of the input), hence input.e is updated with input.l when user clicks ENTER.

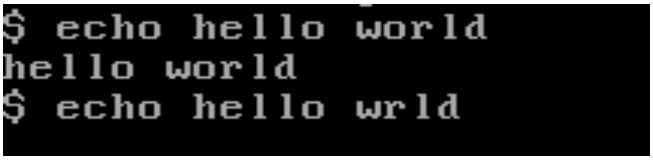

Figure 10: cursor at a middle position - left arrow

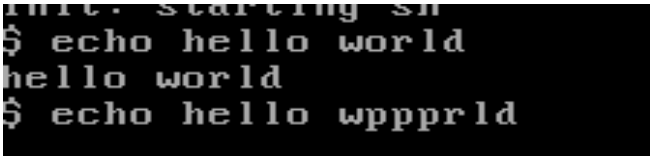

Figure 11: inserted few characters in the middle.

TASK 1.2 - History

historyBuf - structure of history

```
#define MAX_HISTORY 16

struct{
  char buf[MAX_HISTORY][INPUT_BUF]; // to store upto 16 past commands
  int l; // latest command used
  int s; // state of the history, contains the index if up arrow used, else -1
  int nos; // total slots filled in history
} historyBuf;
```

To store the commands that have been previously typed by the user we need some buffer which is named as historyBuf and it contains buffer memory(buf), number of commands that are stored (nos - before user enters 16 commands it is some number below 16, later it is always 16, MIX_HISTORY), s, l are to store state and latest index for navigating through the buffer memory as and when needed. Normally the value of s is reset to -1, when user uses UP arrow it is changed to 0 and we calculate the index of historyBuf to fill the input buffer and console. latest index (l) - state (s) is the index to be used for fetching the previous input, as l is the latest command used and state s is 0 after user clicks UP arrow, l^{th} is fetched first from historyBuf, followed by l-1, l-2 ... until possible.

Storing the latest input in history

```
void saveToHistory(void){
  historyBuf.s = -1;
  if(allSpaces()){
    return;
  }
  if(historyBuf.nos<MAX_HISTORY){
    historyBuf.nos++;
  }
  uint i = 0;
  uint ending = postStrip();
  uint starting = preStrip();
  while(starting + i + input.r<input.l - ending){
    historyBuf.buf[historyBuf.l][i] = input.buf[(starting
    i++;
  }
  historyBuf.buf[historyBuf.l][i] = '\0';
  historyBuf.l = (historyBuf.l+1)%MAX_HISTORY;
}
```

Whenever user inputs return (\r or enter) we need to save the line from input.r to input.l from the input.buf into historyBuf.buf; Also we need to update the index of latest command accordingly. We have used a circular array for implementing historyBuf.buf, so if the number of commands in the history buf is 16 then it automatically erases the oldest command and writes the latest at that spot. saveToHistory() function contains all the instructions to perform above operations. Note that allSpaces() is the function that checks whether the input range(input.r to input.l) contains all spaces and tabs to ensure that blank inputs to be unsaved. Also we need to take out any spaces entered before and after the command the input.buf, preStrip() and postStrip() gives the first and last non-space characters,so that we can iterate between these indices to save them in the history.

Utility functions

Before going further there are some utility functions that are used when user navigates through using UP and DOWN arrows.

1. **clearScreen():** Whenever there is a interrupt of UP or DOWN arrow some input from history is showed on the screen (assume atleast one command is stored in history). If we do not clear the screen

(current command) and the previous command is larger than the command that is coming from the history then there will be some trailing characters hanging in the console. We created this function to put required number of backspaces on the console screen.

```
void clearScreen(void){
  for(int i=input.l;i>input.r;i--){
    consputc(BACKSPACE);
  }
}
```

Figure 12: clearScreen()

```
void copyTypedToTmpbuf(void){
  int j = 0;
  for(int i=input.r;i<input.l;i++,j++){
    tmpbuf[j] = input.buf[i%INPUT_BUF];
  }
  tmpbuf[j] = '\0';
}
```

Figure 13: copyTypedToTmpbuf()

2. **copyTypedToTmpbuf():** As the name suggests it copies input currently typed by the user to an array named tmpbuf. When user is typing some command and suddenly used UP/DOWN to navigate through history, currently typed command should not be thrown away as he might come back and continue writing the same. Hence we are saving this command in a temporary buffer named tmpbuf. It just copies the characters from the input buffer from input.r to input.l into tmpbuf.

3. **fillInputBuffer(int index):** It takes the index of history buffer as parameter and copies the characters from the history buffer at that index into input buffer. This is to ensure that when user navigated through the history and clicked enter the currently shown command has to run (It will run only if it is present in input buffer).

```
void fillInputBuffer(int index){
  uint i = 0;
  char c = historyBuf.buf[index][i];
  while(c!='\0'){
    input.buf[(input.r+i)%INPUT_BUF] = c;
    i++;
    c = historyBuf.buf[index][i];
    input.e++;
    input.l++;
  }
}
```

Figure 14: fillInputBuffer(int)

```
void fillScreenWithHistory(int index){
  uint i = 0;
  char c = historyBuf.buf[index][i];
  while(c!='\0'){
    consputc(c);
    i++;
    c = historyBuf.buf[index][i];
  }
}
```

Figure 15: fillScreenWithHistory(int)

4. **fillScreenWithHistory(int index):** Similar to previous function this method read the history buffer at the given index and prints the characters to the console (using consputc()).
Note that in the both fillScreenWithHistory and fillInputBuffer index is calculated from the state and latest index of the history, will be discussed in UP/DOWN arrow.

UP/DOWN ARROW - things to do

When user clicks UP arrow, there is nothing to do if it is the first command entered by the user(history is empty). Once the user enters some commands into console, he has some commands in his buffer to navigate through. initially historyBuf.s (state variable) is always -1. Once user uses UP Arrow s is increased and index = historyBuf.l - hisotyrBuf.s (latest index of a command - state). As for the first time s is 0, latest entered command is brought to screen and input buffer is filled. Next up arrow increases s further and changes index accordingly to bring the required command. State should always be less than MAX_HISTORY.

- Once UP arrow is used console is cleared using *clearScreen()*.
- Currently entered command (maybe partial) is copied to tmpbuf using *copyTypedToTmpbuf()*.
- Once the tmpbuf filling is over, state and index of the history buffer are changed.
- Input buffer is filled with the characters stored in historyBuf.buf at locaion index using *fillInputBuffer()*.

- console should be filled with the same command that is present in the input buffer to show the user the current text present in the input buffer, which is taken care by *fillScreenWithHistory()*.

```
case UP:
  if(historyBuf.s < historyBuf.nos-1){
    clearScreen();
    if(historyBuf.s == -1){
      copyTypedToTmpbuf();
      // only for the time before accessing the history store the contents
      // of buffer in tmpbuf.
    }
    historyBuf.s++; // change the state by 1. to access the next command in the histo
    // index of history is relatively history.s from reference as history.l
    // history.l - history.s - 1
    input.l = input.r, input.e = input.r;
    int hisBufIndex = (MAX_HISTORY +  historyBuf.l - historyBuf.s - 1)%MAX_HISTORY;
    fillInputBuffer(hisBufIndex);
    fillScreenWithHistory(hisBufIndex);
  }// nothing to do elsewise
  break;
```

Figure 16: what happens if user clicks UP arrow.

Similar to UP arrow DOWN arrow also might have to use history buffer to bring the input stored in the next latest input stored in the history buffer to the current input if UP arrow is used atleast once. If UP arrow is not used or both arrows are used for same time (state is -1), there is nothing to do. Note that if state is 0 and DOWN arrow is used we have to fill input buffer with the contents of tmpbuf instead of historyBuf.buf[index] as the user want to further use the previously typed text in the console. If state is greater than 0, the process of filling the console and input buffer is same as that of for UP arrow case, else the inputs are brought from tmpbuf as shown in the figure below.

```
case DOWN:
  if(historyBuf.s > 0){// when up arrow is multiple times
    clearScreen();
    historyBuf.s--; // to bring the previous state
    input.l = input.r, input.e = input.r;
    int hisBufIndex = (MAX_HISTORY + historyBuf.l - historyBuf.s -1)%MAX_HI
    fillInputBuffer(hisBufIndex);
    fillScreenWithHistory(hisBufIndex);
  }else if(historyBuf.s==0){
    // when up arrow is used once we want to bring back the command we type
    clearScreen();
    historyBuf.s--;
    // bringing tmpbuf contents to input buffer and scree.
    input.l = input.r, input.e = input.r;
    uint len = 0;
    char c = tmpbuf[len];
    while(c!='\0'){
      input.buf[(input.r+len)%INPUT_BUF] = c;
      consputc(c);
      len++;
      c = tmpbuf[len];
      input.e++;
      input.l++;
    }
  }// nothing to do if state is -1 as up arrow is not used at all.
  break;
```

Figure 17: what happens if user clicks DOWN arrow.

History

History system call is created as per the recommendations and in the main function of sh.c whenever hitory is read in the buffer it calls this fucntion to get the data and prints it to the terminal. syscall.c, syscall.h, sysproc.c user.h, usys.S have been modified to facilitate this system call. The output of this command is shown below as an example.

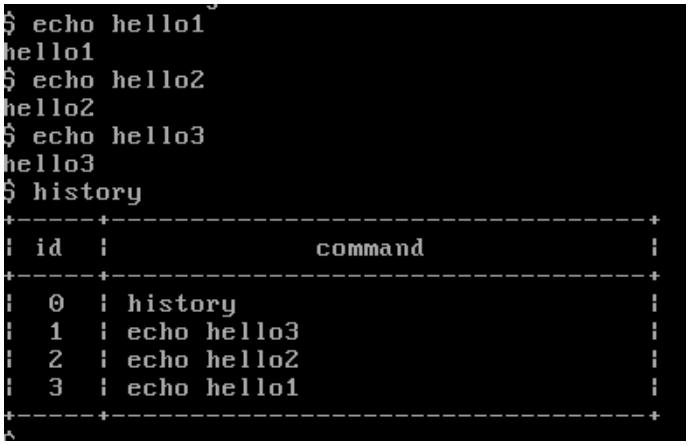

Figure 18: output of history command

TASK 2

In this task we have been extending the struct proc so that it is useful in future for evaluating the performances of each scheduling policies.

- **proc.h:** First of all we updated the proc.h file where we added ctime, retime, rutime and stime inside the proc struct. After that we declared the functions statisitics_handler(to update the process time values depending upon it's state) and wait2(extends the wait system call) inside the same file.
- **proc.c:** Then we updated the proc.c file where we implemented the wait2 and statistics_handler functions. Also we updated the allocproc function.
  - **wait2:**int wait2(int* retime,int *rutime, int*stime):-This function takes 3 arguments i.e. retime, rutime and stime and updates these values inside it.It returns the PID of the successfully terminated child process. If there is no process to terminate then it will return 1. In this function for each process in the process table we check whether the current running process is parent of this process. If it is then we are checking whether the process isin ZOMBIE state or not. If it is then we are assigning corresponding time values inside the arguments taken and terminating that process and then finally return the PID of the process. If we don't find any such process we return 1.
  - **statistics_handler:**void statistics_handler() :-In this function we will iterate through each process present inside the process table and will update the statistics for the process according to the state(RUNNING, SLEEPING, RUNNABLE) in which it is present.

```
void statistics_handler()
{
  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:
      ;
    }
    // cprintf("%d %d %d\n",p->stime,p->retime,p->rutime);
  }
  release(&ptable.lock);
}
```

Figure 19: statistics_handler()

```
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

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

  release(&ptable.lock);
```

Figure 20: reset the variable in allocproc()

  - **alloc_proc:**static struct proc* allocproc(void): When a new process has been created it is created by allocproc so here as a new process is getting created therefore we are assigning the creation time ctime equal to ticks and intialising the retime,rutime, stime as zero.

- **trap.c:**Then we updated the trap.c file. Inside the trap function we called the statistics_handler inside the switch in the case where tf-> trapno is equal to T_IRQ +IRQ_TIMER. Whenever ticks gets updated i.e. incremented we are also calling the statistics_handler that time which will update all the processes available.

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }

  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      statistics_handler();
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
```

Figure 21: called statistics_handler() from trap()

- **wait2 system call:**Then we created a system call which will be used to call the wait2 function inside the sysproc.c file. In this we are first defining 3 integer pointer variables and placing values inside them (the one received from the user). Then we are calling the wait2 function and then returning the value wait2 returned.

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

Figure 22: sysproc.c system call for wait2()

After that we defined a macro for this function and gave it the system call number 22 inside thesyscall.h file and this helped us to connect it to the implemented function.

Then to make this function visible to the entire program I added the line extern int sys_wait2(void) inside the syscall.c file . Then inside the syscalls array we added sys_wait2 at 22nd position.

Then to connect the user's call to the system call function we created a user level system call definition inside the usys.S file.

Then comes the final part, inside the user.h (user header file) I added the user call int wait2(int*, int*, int*) . and this user call is the one which takes 3 arguments retime, rutime,stime and updates them and then finally returns the PID of the terminated child process.

- **wait2_test:** Then we created a user level test file named wait2_test.c which calls this function and prints the respective information. In this file we are first creating new duplicate child processes by using fork() and then calling wait2 and then storing it's return value in the pid variable. Then if the fork returns 0 value then it means that the current process is a child process. Therefore for that iteration we are just printing "Child Process" in the console. If it is not zero then we will check whether pid==fork's return value; if it so then we will print the retime,rutime and stime for the terminated child process.Else we will print the error that is not able to terminate the child process on the console.

  To make this program available for xv6 source code for compilation we included wait2_test in the user programs or UPROGS by adding the line _wait2_test inside the UPROGS present inside the Makefile.Then we reloaded the QEMU terminal by executing following commands:-
  1.) make clean
  2.) make qemu
  These commands will clean, compile and link things respectively.Then I executed the wait2_test to get the desired output. We can also list all the commands inside fs.img using ls command through which we ensure that the wait2_test command is available.

```c
int main()
{
    int retime,rutime,stime,n1;
    int k = 0;
    for (int a = 0; a < 5; a++)
    {
        n1 = fork();


        int pid = wait2(&retime, &rutime, &stime);
        if (!n1){
            k++; // k denote the number of ancestors
                 // of the current child process
            printf(1, "%d Child Process.\n", k);
            continue;
        }else if (pid == n1){…
        // print(pid, retime, rutime, stime);
        else{
            printf(1, "Not able to terminate the child process.\n");
        }
    }
    exit();
}
```

Figure 23: wait2_test.c

- **Test_case Explanation:** The below picture depicts the output of the file. As In the main function we have called fork() 3 times which means In total this will create 7 child processes recursively for which there are no further child processes to terminate. In these cases we have printed "Child Process." in the console.

  Further When we are in a process which has child processes We print corresponding values of PID, retime, rutime, stime respectively for the terminated child processes.So here we can see we have terminated 7 child processes successfully and printed their statistics.

```
$ wait2_test
Child Process.
Child Process.
Child Process.
| PID: 6 | Retime: 0 | Rutime: 0 | Stime: 0 |
| PID: 5 | Retime: 0 | Rutime: 3 | Stime: 0 |
Child Process.
| PID: 7 | Retime: 0 | Rutime: 1 | Stime: 0 |
| PID: 4 | Retime: 0 | Rutime: 4 | Stime: 4 |
Child Process.
Child Process.
| PID: 9 | Retime: 0 | Rutime: 1 | Stime: 0 |
| PID: 8 | Retime: 0 | Rutime: 4 | Stime: 1 |
Child Process.
| PID: 10 | Retime: 0 | Rutime: 1 | Stime: 0 |
$
```

Figure 24: output for wait2_test command