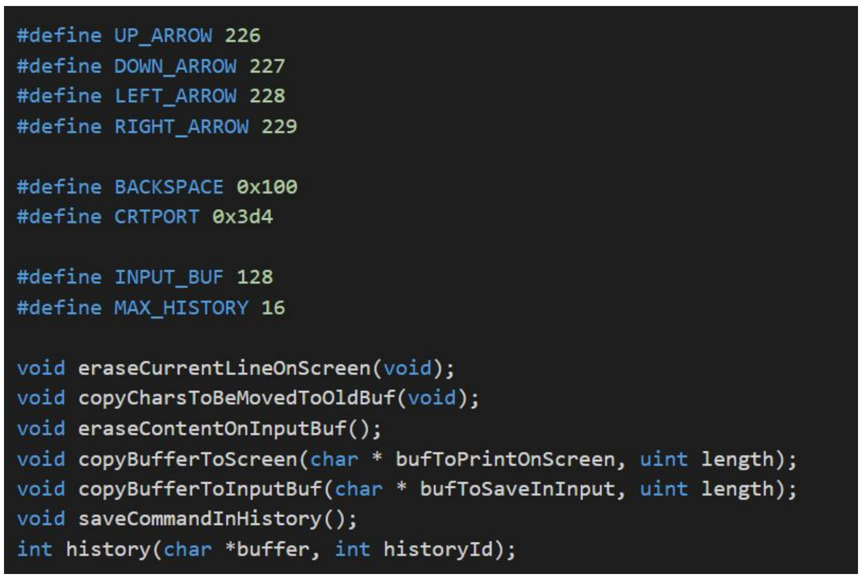
CS344 OS Lab
Assignment - 2A

Group 1:
Eshan Trehan - 190123070
Anuraag Mahajan - 190101110
Kshitij Singhal - 190123032
Hardik Suhag - 190101038

TASK 1: Improving the console

Caret navigation and Shell History Ring

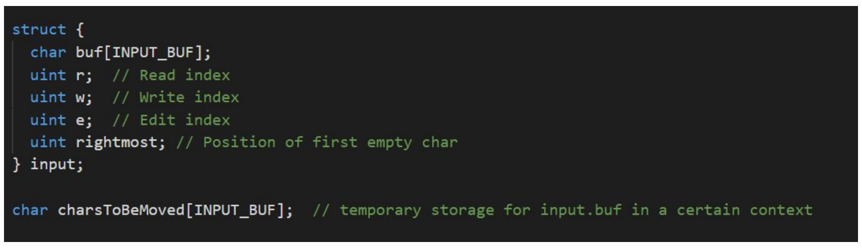
We start with console.c which handles the input and output.



```
#define UP_ARROW 226
#define DOWN_ARROW 227
#define LEFT_ARROW 228
#define RIGHT_ARROW 229

#define BACKSPACE 0x100
#define CRTPORT 0x3d4

#define INPUT_BUF 128
#define MAX_HISTORY 16

void eraseCurrentLineOnScreen(void);
void copyCharsToBeMovedToOldBuf(void);
void eraseContentOnInputBuf();
void copyBufferToScreen(char * bufToPrintOnScreen, uint length);
void copyBufferToInputBuf(char * bufToSaveInInput, uint length);
void saveCommandInHistory();
int history(char *buffer, int historyId);
```

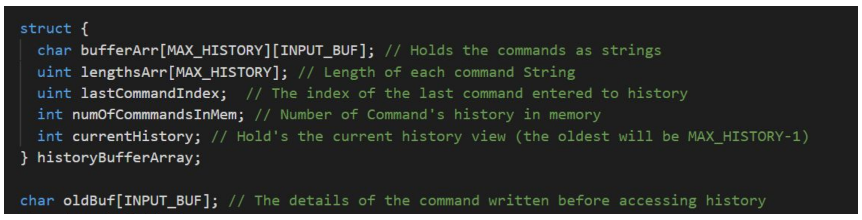
1) Defining MACROS for
- important keystrokes used in caret navigation
- MAX_HISTORY (specifying no. of commands)
- buffer size for holding history

2) Helper function prototypes.



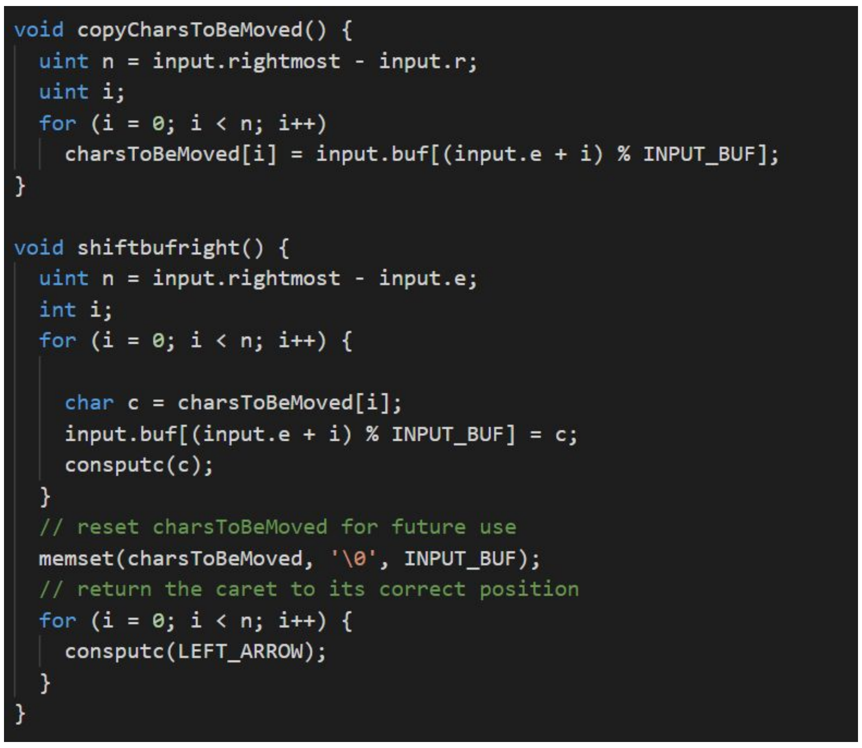
```
struct {
  char buf[INPUT_BUF];
  uint r;  // Read index
  uint w;  // Write index
  uint e;  // Edit index
  uint rightmost; // Position of first empty char
} input;

char charsToBeMoved[INPUT_BUF];  // temporary storage for input.buf in a certain context
```

Addition of rightmost index pointer to mark the end of line and a buffer to store the current line, required to move the contents of insertion or deletion.



```
struct {
  char bufferArr[MAX_HISTORY][INPUT_BUF]; // Holds the commands as strings
  uint lengthsArr[MAX_HISTORY]; // Length of each command String
  uint lastCommandIndex;  // The index of the last command entered to history
  int numOfCommmandsInMem; // Number of Command's history in memory
  int currentHistory; // Hold's the current history view (the oldest will be MAX_HISTORY-1)
} historyBufferArray;

char oldBuf[INPUT_BUF]; // The details of the command written before accessing history
```

Defining the custom data structure for storing and accessing commands history.

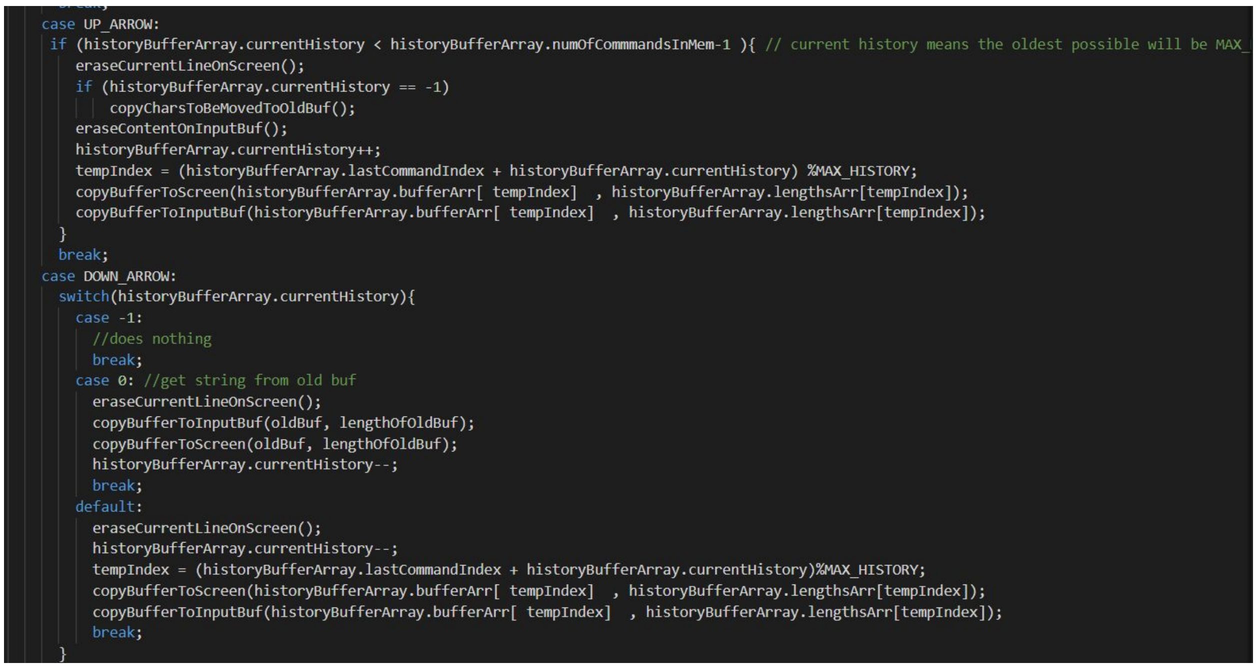


```
void copyCharsToBeMoved() {
  uint n = input.rightmost - input.r;
  uint i;
  for (i = 0; i < n; i++)
    charsToBeMoved[i] = input.buf[(input.e + i) % INPUT_BUF];
}

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

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

CopyCharsToBeMoved():
Copy input.buf to a safe location. Used only when punching in new keys and the caret isn't at the end of the line.

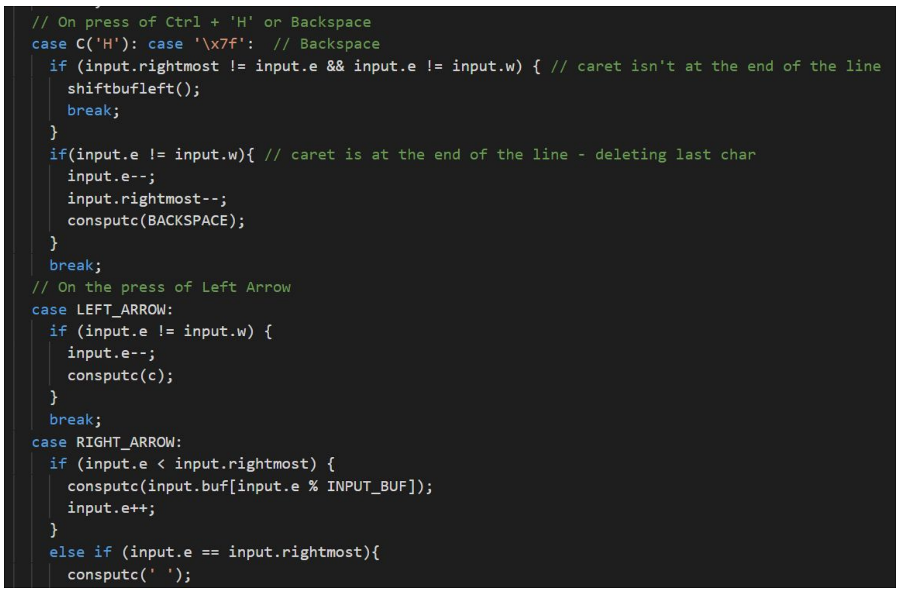
Shiftbufright():
Shift input.buf one byte to the right, and repaint the chars on-screen. Used only when punching in new keys and the caret isn't at the end of the line.

Similarly for left keystroke.



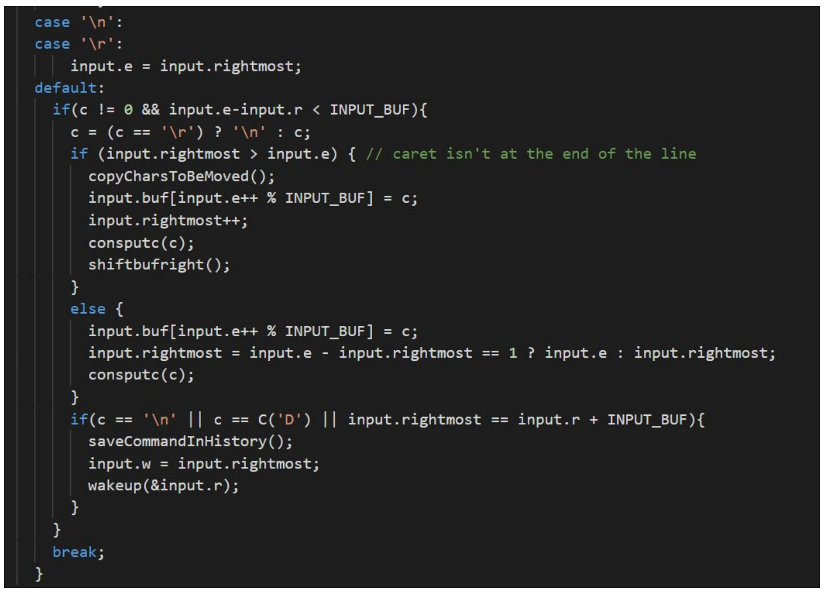
```
case UP_ARROW:
 if (historyBufferArray.currentHistory < historyBufferArray.numOfCommmandsInMem-1 ){ // current history means the oldest possible will be MAX_
   eraseCurrentLineOnScreen();
   if (historyBufferArray.currentHistory == -1)
     copyCharsToBeMovedToOldBuf();
   eraseContentOnInputBuf();
   historyBufferArray.currentHistory++;
   tempIndex = (historyBufferArray.lastCommandIndex + historyBufferArray.currentHistory) %MAX_HISTORY;
   copyBufferToScreen(historyBufferArray.bufferArr[ tempIndex]  , historyBufferArray.lengthsArr[tempIndex]);
   copyBufferToInputBuf(historyBufferArray.bufferArr[ tempIndex]  , historyBufferArray.lengthsArr[tempIndex]);
  }
  break;
case DOWN_ARROW:
  switch(historyBufferArray.currentHistory){
   case -1:
    //does nothing
    break;
   case 0: //get string from old buf
    eraseCurrentLineOnScreen();
    copyBufferToInputBuf(oldBuf, lengthOfOldBuf);
    copyBufferToScreen(oldBuf, lengthOfOldBuf);
    historyBufferArray.currentHistory--;
    break;
   default:
    eraseCurrentLineOnScreen();
    historyBufferArray.currentHistory--;
    tempIndex = (historyBufferArray.lastCommandIndex + historyBufferArray.currentHistory)%MAX_HISTORY;
    copyBufferToScreen(historyBufferArray.bufferArr[ tempIndex]  , historyBufferArray.lengthsArr[tempIndex]);
    copyBufferToInputBuf(historyBufferArray.bufferArr[ tempIndex]  , historyBufferArray.lengthsArr[tempIndex]);
    break;
  }
```

Handling the retrieval of the next / last item in the history respectively using switch case for UP_ARROW and DOWN_ARROW



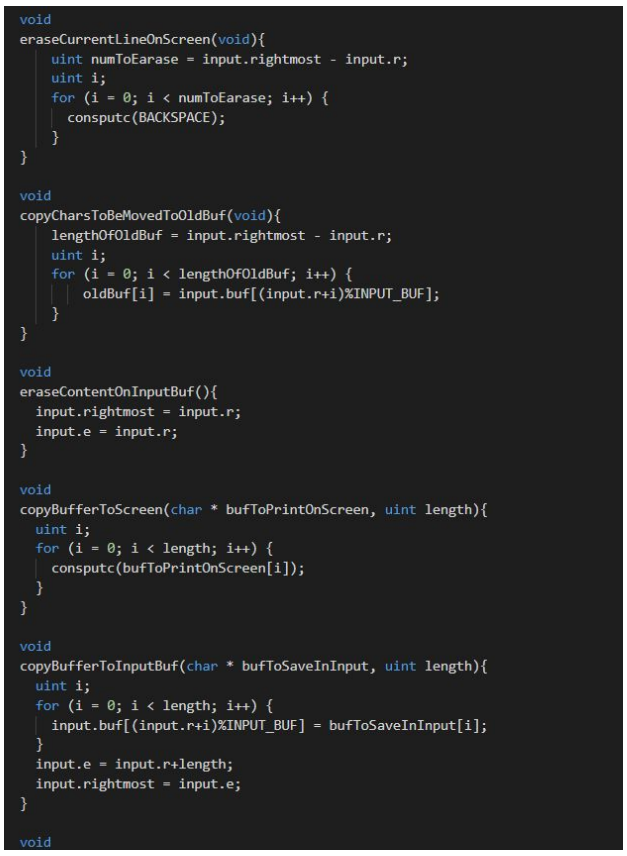
```
// On press of Ctrl + 'H' or Backspace
case C('H'): case '\x7f':  // Backspace
  if (input.rightmost != input.e && input.e != input.w) { // caret isn't at the end of the line
    shiftbufleft();
    break;
  }
  if(input.e != input.w){ // caret is at the end of the line - deleting last char
    input.e--;
    input.rightmost--;
    consputc(BACKSPACE);
  }
  break;
// On the press of Left Arrow
case LEFT_ARROW:
  if (input.e != input.w) {
    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){
    consputc(' ');
```

Handling left/right/backspace caret navigation

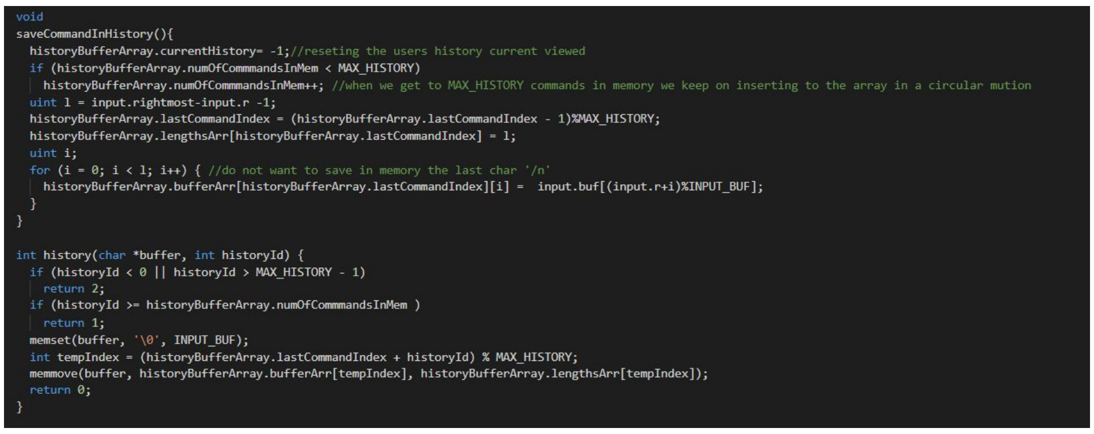


```
case '\n':
case '\r':
    input.e = input.rightmost;
default:
  if(c != 0 && input.e-input.r < INPUT_BUF){
    c = (c == '\r') ? '\n' : c;
    if (input.rightmost > input.e) { // caret isn't at the end of the line
      copyCharsToBeMoved();
      input.buf[input.e++ % INPUT_BUF] = c;
      input.rightmost++;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

Handling next line and insertion



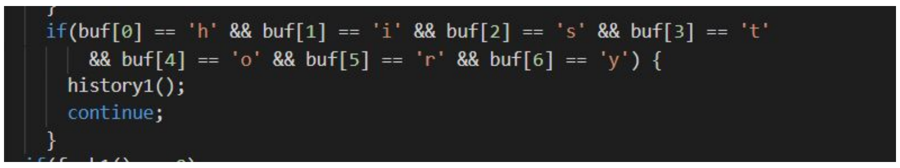
```
void
eraseCurrentLineOnScreen(void){
    uint numToEarase = input.rightmost - input.r;
    uint i;
    for (i = 0; i < numToEarase; i++) {
      consputc(BACKSPACE);
    }
}

void
copyCharsToBeMovedToOldBuf(void){
    lengthOfOldBuf = input.rightmost - input.r;
    uint i;
    for (i = 0; i < lengthOfOldBuf; i++) {
      oldBuf[i] = input.buf[(input.r+i)%INPUT_BUF];
    }
}

void
eraseContentOnInputBuf(){
  input.rightmost = input.r;
  input.e = input.r;
}

void
copyBufferToScreen(char * bufToPrintOnScreen, uint length){
  uint i;
  for (i = 0; i < length; i++) {
    consputc(bufToPrintOnScreen[i]);
  }
}

void
copyBufferToInputBuf(char * bufToSaveInInput, uint length){
  uint i;
  for (i = 0; i < length; i++) {
    input.buf[(input.r+i)%INPUT_BUF] = bufToSaveInInput[i];
  }
  input.e = input.r+length;
  input.rightmost = input.e;
}

void
```



```
void
saveCommandInHistory(){
  historyBufferArray.currentHistory= -1;//reseting the users history current viewed
  if (historyBufferArray.numOfCommmandsInMem < MAX_HISTORY)
    historyBufferArray.numOfCommmandsInMem++; //when we get to MAX_HISTORY commands in memory we keep on inserting to the array in a circular mution
  uint l = input.rightmost-input.r -1;
  historyBufferArray.lastCommandIndex = (historyBufferArray.lastCommandIndex - 1)%MAX_HISTORY;
  historyBufferArray.lengthsArr[historyBufferArray.lastCommandIndex] = l;
  uint i;
  for (i = 0; i < l; i++) { //do not want to save in memory the last char '/n'
    historyBufferArray.bufferArr[historyBufferArray.lastCommandIndex][i] =  input.buf[(input.r+i)%INPUT_BUF];
  }
}

int history(char *buffer, int historyId) {
  if (historyId < 0 || historyId > MAX_HISTORY - 1)
    return 2;
  if (historyId >= historyBufferArray.numOfCommmandsInMem )
    return 1;
  memset(buffer, '\0', INPUT_BUF);
  int tempIndex = (historyBufferArray.lastCommandIndex + historyId) % MAX_HISTORY;
  memmove(buffer, historyBufferArray.bufferArr[tempIndex], historyBufferArray.lengthsArr[tempIndex]);
  return 0;
}
```

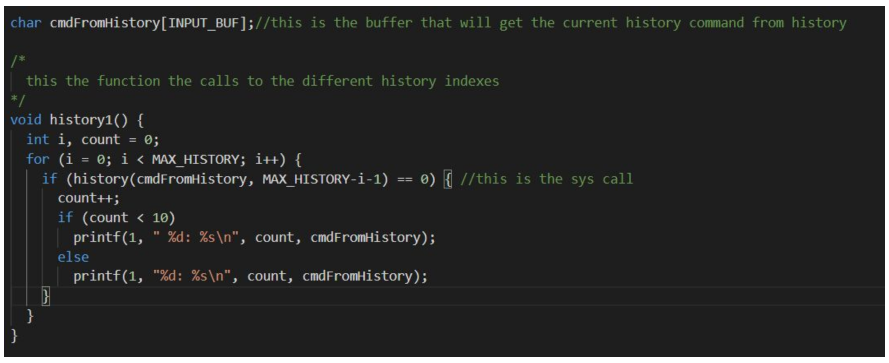
Defining various small helper functions used for execution of shell history and the history system call :
int history (char * buffer, int historyId)

NOTE:
While saving commands (saveCommandInHistory()) if number of commands exceed max size, we store the commands in a cyclic manner inserting to index zero on every overflow.



```
  if(buf[0] == 'h' && buf[1] == 'i' && buf[2] == 's' && buf[3] == 't'
     && buf[4] == 'o' && buf[5] == 'r' && buf[6] == 'y') {
    history1();
    continue;
  }
```

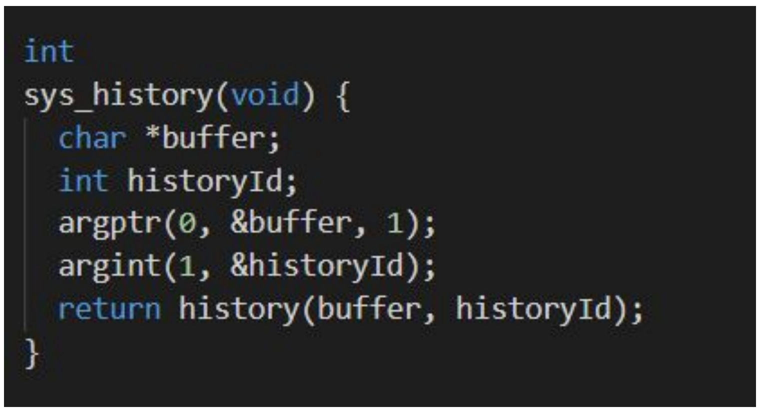
sh.c
Execution of history command in shell source script main(), so that upon writing the command a full list of the history should be printed to screen.



```
char cmdFromHistory[INPUT_BUF];//this is the buffer that will get the current history command from history

/*
  this the function the calls to the different history indexes
*/
void history1() {
  int i, count = 0;
  for (i = 0; i < MAX_HISTORY; i++) {
    if (history(cmdFromHistory, MAX_HISTORY-i-1) == 0) { //this is the sys call
      count++;
      if (count < 10)
        printf(1, " %d: %s\n", count, cmdFromHistory);
      else
        printf(1, "%d: %s\n", count, cmdFromHistory);
    }
  }
}
```

sh.c
Helper function that traverses through the history buffer, printing each command on a new line.

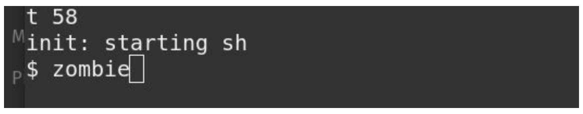
Finally, to add the history system call the following files are edited as in Assignment 1 -
syscall.c
syscall.h
sysproc.c
user.h
usys.S



```
int
sys_history(void) {
  char *buffer;
  int historyId;
  argptr(0, &buffer, 1);
  argint(1, &historyId);
  return history(buffer, historyId);
}
```

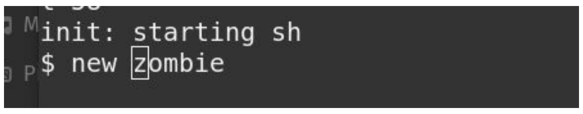
Adding the real implementation to sysproc.c

Output



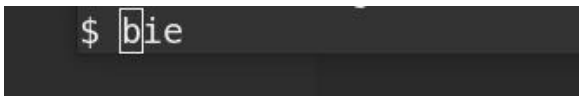
```
t 58
init: starting sh
$ zombie
```

Caret Navigation:



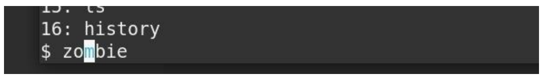
```
init: starting sh
$ new zombie
```

Character insertion from middle

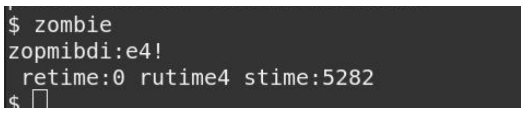


```
$ bie
```

Character deletion from middle



```
15: ls
16: history
$ zombie
```



```
$ zombie
zopmibdi:e4!
 retime:0 rutime4 stime:5282
$ 
```

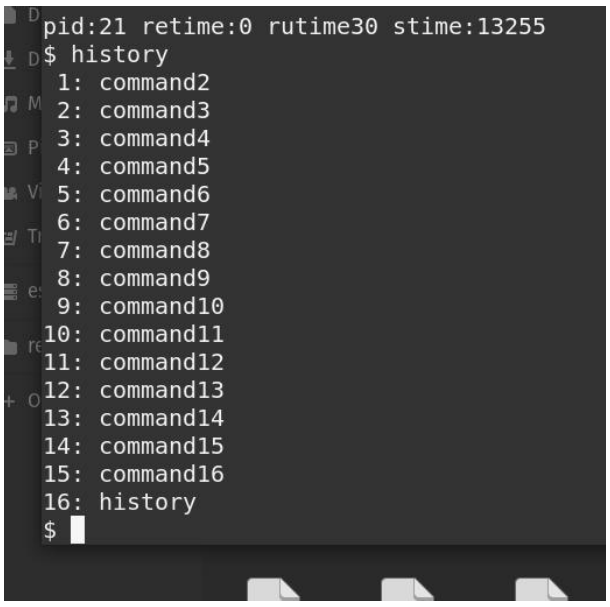
Execution from middle.

***Note** : End line results in moving to the next line no matter where the caret is*

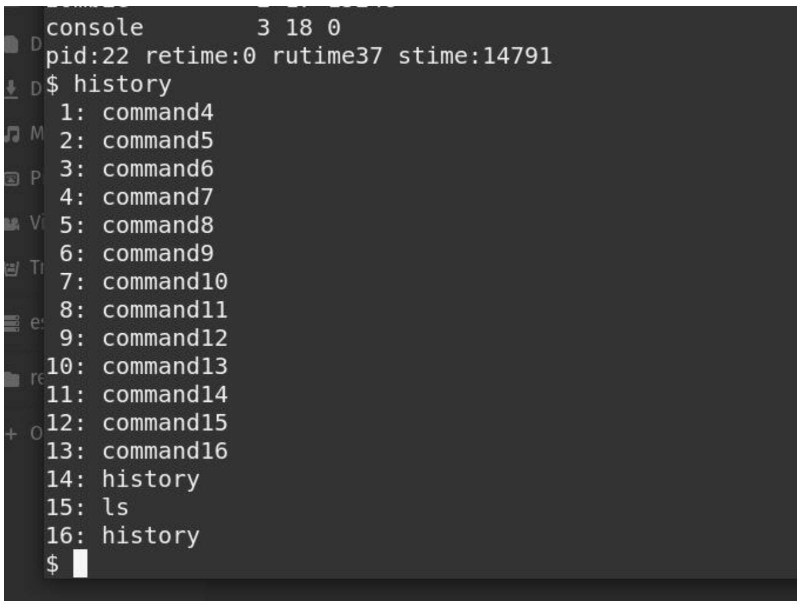
History:

Up and down arrows lead to display of previous and next command respectively

History command displays the commands in a list



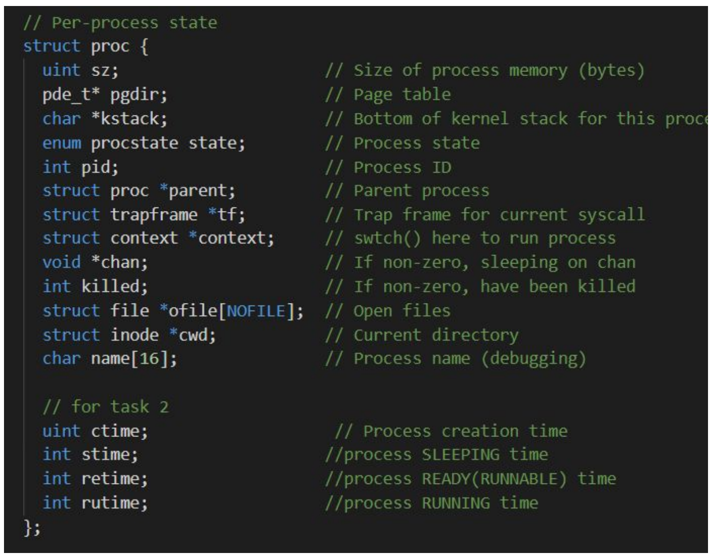
```
pid:21 retime:0 rutime30 stime:13255
$ history
 1: command2
 2: command3
 3: command4
 4: command5
 5: command6
 6: command7
 7: command8
 8: command9
 9: command10
10: command11
11: command12
12: command13
13: command14
14: command15
15: command16
16: history
$ 
```



```
console        3 18 0
pid:22 retime:0 rutime37 stime:14791
$ history
 1: command4
 2: command5
 3: command6
 4: command7
 5: command8
 6: command9
 7: command10
 8: command11
 9: command12
10: command13
11: command14
12: command15
13: command16
14: history
15: ls
16: history
$ 
```

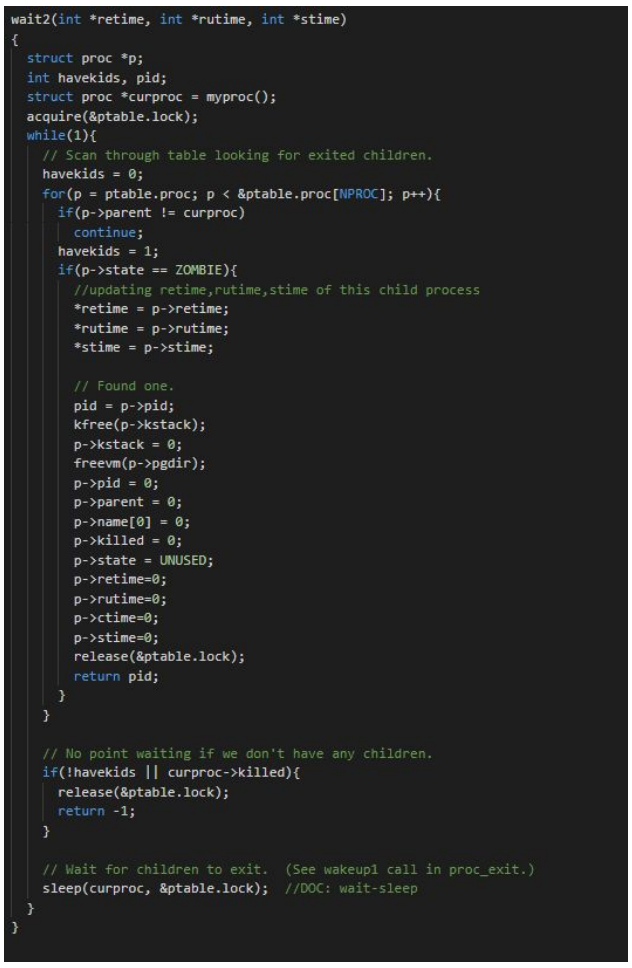
Boundary Case: Cyclic insertion of commands in case of overflow (>MAX_HISTORY)

TASK 2: Statistics



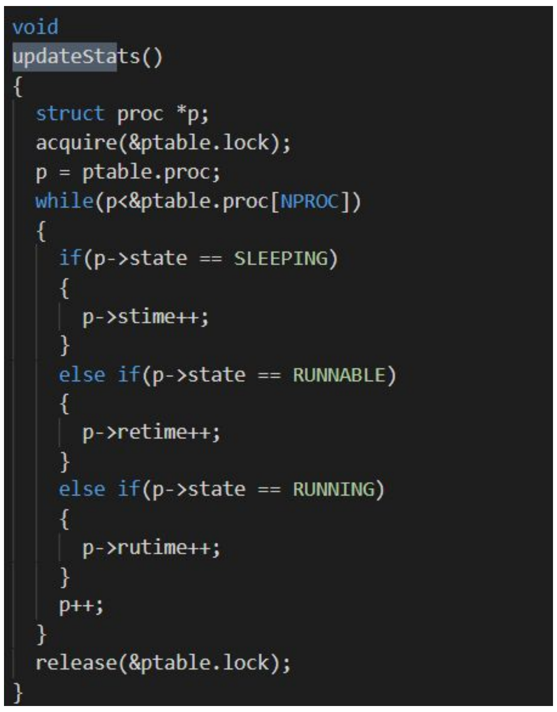
```c
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this proc
  enum procstate state;        // Process state
  int pid;                     // 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)

  // for task 2
  uint ctime;                   // Process creation time
  int stime;                   //process SLEEPING time
  int retime;                  //process READY(RUNNABLE) time
  int rutime;                  //process RUNNING time
};
```

proc.h
Extending the struct proc to add the required parameters.



```c
wait2(int *retime, int *rutime, int *stime)
{
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc();
  acquire(&ptable.lock);
  while(1){
    // 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){
        //updating retime,rutime,stime of this child process
        *retime = p->retime;
        *rutime = p->rutime;
        *stime = p->stime;

        // Found one.
        pid = p->pid;
        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;
        p->retime=0;
        p->rutime=0;
        p->ctime=0;
        p->stime=0;
        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
  }
}
```

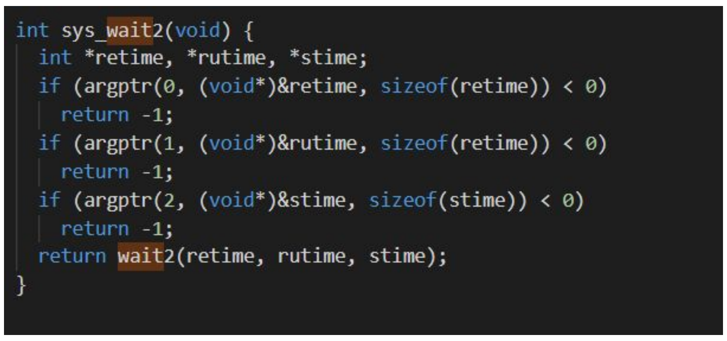
proc.c
Implementation of wait2() to fetch and assign the required values. (similar to wait() func)



```c
void
updateStats()
{
  struct proc *p;
  acquire(&ptable.lock);
  p = ptable.proc;
  while(p<&ptable.proc[NPROC])
  {
    if(p->state == SLEEPING)
    {
      p->stime++;
    }
    else if(p->state == RUNNABLE)
    {
      p->retime++;
    }
    else if(p->state == RUNNING)
    {
      p->rutime++;
    }
    p++;
  }
  release(&ptable.lock);
}
```

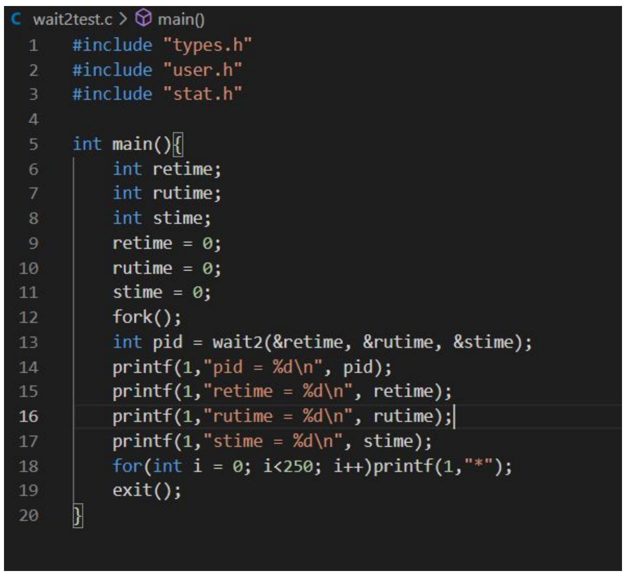
proc.c
updateStats(): this method will run every clock tick and update the statistic fields for each proc

Standard protocol followed for adding the system call, editing the following files -
syscall.c
syscall.h
sysproc.c
user.h
usys.S
Makefile (for user program)



```c
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);
}
```

sys_wait2() in sysproc.c which calls wait2() system function

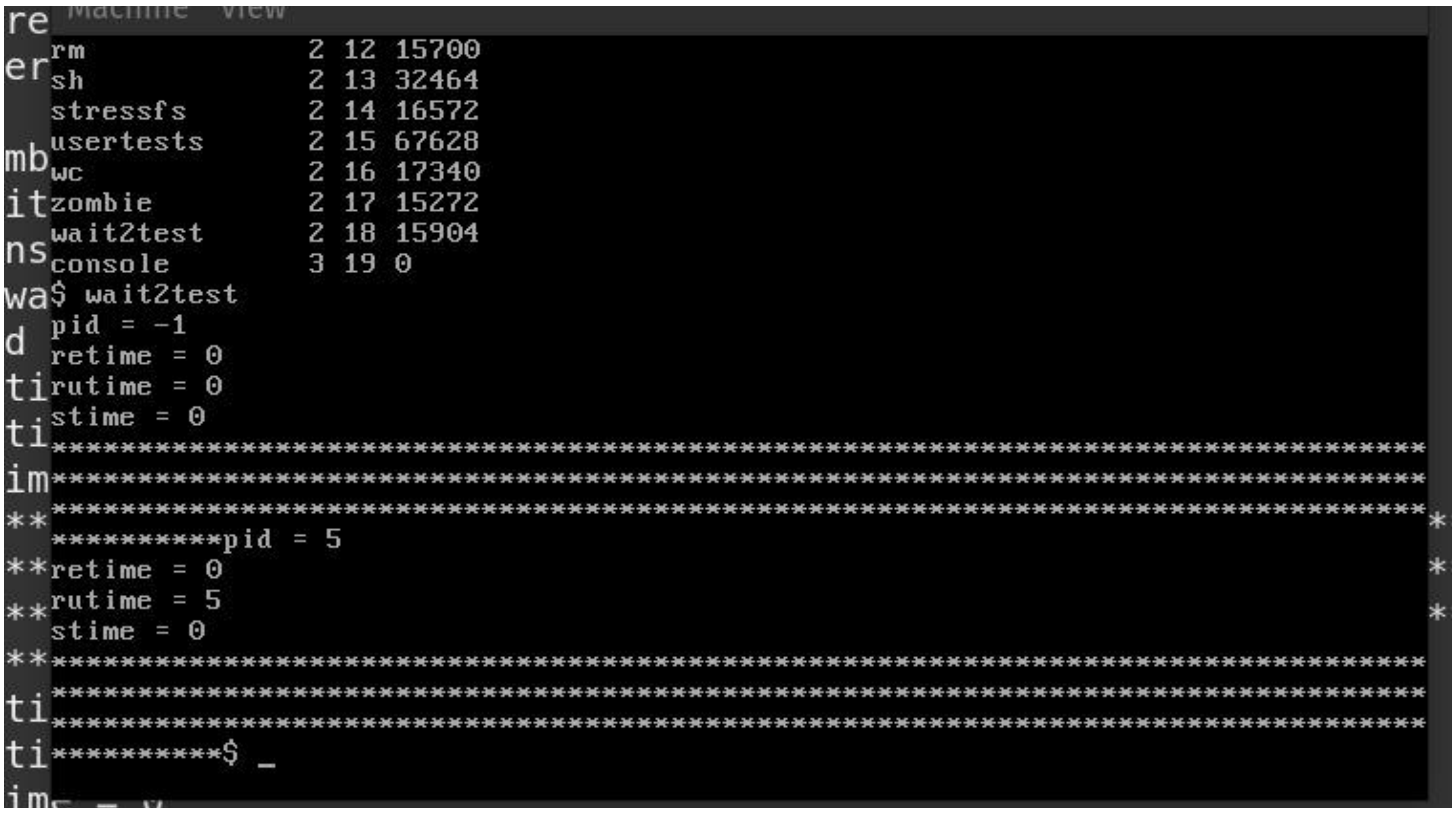


```c
#include "types.h"
#include "user.h"
#include "stat.h"

int main(){
    int retime;
    int rutime;
    int stime;
    retime = 0;
    rutime = 0;
    stime = 0;
    fork();
    int pid = wait2(&retime, &rutime, &stime);
    printf(1,"pid = %d\n", pid);
    printf(1,"retime = %d\n", retime);
    printf(1,"rutime = %d\n", rutime);
    printf(1,"stime = %d\n", stime);
    for(int i = 0; i<250; i++)printf(1,"*");
    exit();
}
```

wait2test.c

User program to test the working of the wait2() function which calls this function and print the respective information.

Output



```
rm             2 12 15700
sh             2 13 32464
stressfs       2 14 16572
usertests      2 15 67628
wc             2 16 17340
zombie         2 17 15272
wait2test      2 18 15904
console        3 19 0
$ wait2test
pid = -1
retime = 0
rutime = 0
stime = 0
********************************************************************************
********************************************************************************
********************************************************************************
**********pid = 5
retime = 0
rutime = 5
stime = 0
********************************************************************************
********************************************************************************
********************************************************************************
**********$ _
```

We can see two wait2() calls, corresponding to child and parent process due to forking.
Note: The asterisk are printed to add a reasonable delay.