In God we trust

Os lab EX1

Mani Hosseini 810102552
Shayan Maleki 810102515
Hamid Mahmoudi 810102549

Introduction

1) The main three jobs of an operating system are:

1-1) One of the jobs that an OS has is multiplexing, which is to share resources like cpu among multiple programs.
1-2) Another job it has is Abstraction. It means it simplifies the hardware by hiding complex details of a hardware from programs and provides simple and universal commands instead.
1-3) And another main job it has is isolation and interaction. This means it provides a controlled way for programs to use a shared data and at the same time ensures they are isolated from each other so one program doesn't harm another one from bugs or crashes.

2) No,an OS is not needed everywhere. A key requirement for an operating system is to support several activities at once.So simple devices that do only a single dedicated task don't need an operating system. An OS becomes essential when a system must handle multiple activities and the OS provides isolation between them as well as handling shared data access and sharing hardware resources.

3) The xv6 OS is implemented as a monolithic kernel as all the main parts of this OS are packed together into one big program that runs with full power over the computer.
Also it's worth mentioning that it's both simpler for people writing the OS as everything is in the same place and it is better for learning.

4) It is a multitasking system. It can run multiple programs at the same time. It can quickly switch CPU's attention between different programs by letting one run for a short time and pausing it.

5) A program is a passive file and it's just a list of instructions and data sitting there and not doing anything.

But a process is the active instance created by the OS after it gets loaded into memory and its instructions are ready to execute.

6)

6-1:
A process in xv6 is built from a few key components:

User Memory: This is the program's own space, containing its code, data, heap, and stack.

Kernel Stack: A private stack used when the process executes inside the kernel during a system call or interrupt.

Page Table: The map that translates the process's virtual addresses to physical memory addresses, providing memory isolation.

Process State: A value that tracks whether the process is running, waiting, ready, etc.

Process ID (PID): A unique number to identify the process.

Saved Context: A saved copy of the CPU registers, allowing the process to be paused and later resumed from the exact same point.

6-2:
xv6 uses a simple round-robin scheduling method:

Pause: The OS stops the current process and saves its CPU state in its context.

Choose: The scheduler picks the next ready-to-run process from the list.

Resume: The OS loads the saved context of the new process, which starts running exactly where it left off.

And repeats this method over and over.

7) A file descriptor is a small integer that serves as a handle or reference to an open file, pipe, or device. It provides a unified interface for I/O operations; allowing programs to

read from and write to different types of resources using the same set of system calls such as read, write and close.

A pipe is a temporary, one-way communication channel that connects processes, allowing the output of one process to become the input of another.

In xv6 it works as followed:

Creation: The operating system creates a pipe, which automatically provides two ends: one for writing data and one for reading it.

Communication: After a process creates a child process, both the parent and child inherit access to the pipe. One process can send data by writing to the "write end," and the other can receive that data by reading from the "read end."

Synchronization: The operating system manages the pipe intelligently. If a process tries to read from an empty pipe, it is put to sleep until data arrives. Similarly, if a process writes to a full pipe, it waits until there is space.

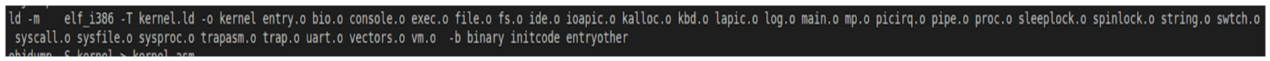
8) The fork system call creates a new child process that is the exact duplicate of the parent process, copying its file description and memory.

The exec however replaces the current processes memory and instructions with a new program loaded from a file, but it preserves the original processes file descriptor file.

The advantage of separating fork and exec is that it allows a parent process to modify the child process's environment before the new program runs. This separation creates a crucial setup phase, most notably used by the shell to manipulate the child's file descriptors to implement I/O redirection. The new program can then be executed without needing any special code to handle the changes.

Setup

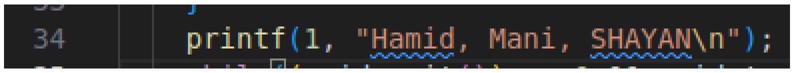
9)The final xv6 kernel is built by the linker command that combines all compiled object files (.o) into a single executable called `kernel`. This is the file that QEMU uses to run the xv6 operating system. All previous compilation steps only generate intermediate object files.



```
ld -m    elf_i386 -T kernel.ld -o kernel entry.o bio.o console.o exec.o file.o fs.o ide.o ioapic.o kalloc.o kbd.o lapic.o log.o main.o mp.o picirq.o pipe.o proc.o sleeplock.o spinlock.o string.o swtch.o
 syscall.o sysfile.o sysproc.o trapasm.o trap.o uart.o vectors.o vm.o  -b binary initcode entryother
```

Boot Message

By adding a print line to our init.c, we can print our names after a successful boot and moving to user mode.



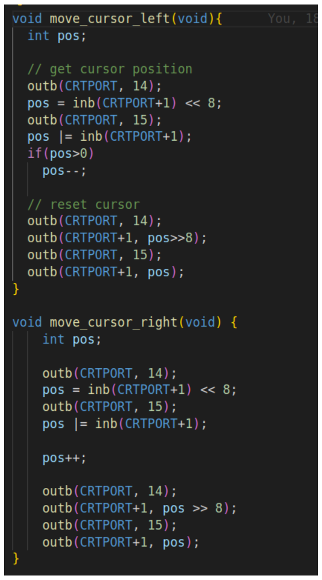
```c
printf(1, "Hamid, Mani, SHAYAN\n");
```

Adding new features

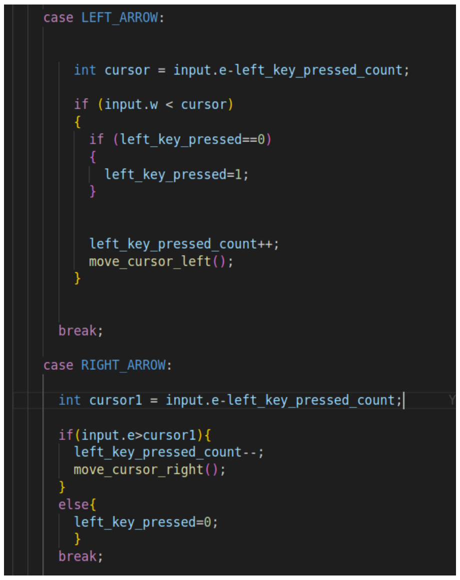
1-cursor Movement

● Left and right arrows:

New cases are added to console interrupt which moves the cursor. Also move cursor left and right were added to update the the arrows visually.

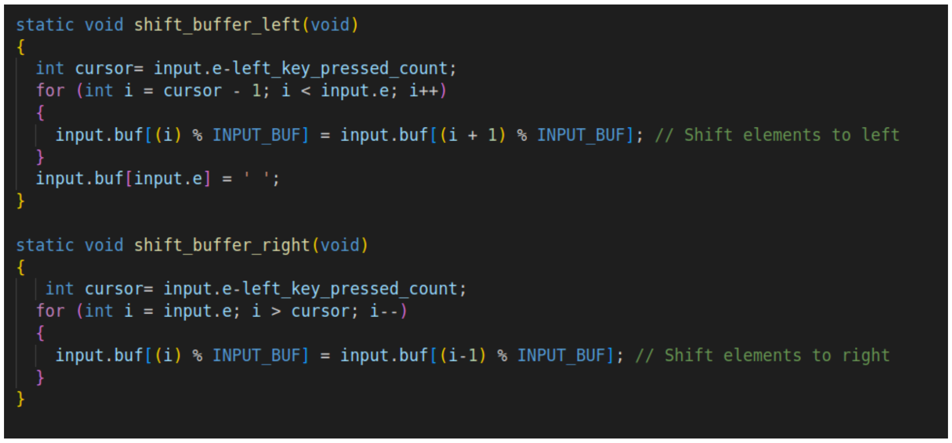


```c
void move_cursor_left(void){
  int pos;

  // get cursor position
  outb(CRTPORT, 14);
  pos = inb(CRTPORT+1) << 8;
  outb(CRTPORT, 15);
  pos |= inb(CRTPORT+1);
  if(pos>0)
    pos--;

  // reset cursor
  outb(CRTPORT, 14);
  outb(CRTPORT+1, pos>>8);
  outb(CRTPORT, 15);
  outb(CRTPORT+1, pos);
}

void move_cursor_right(void) {
    int pos;

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

    pos++;

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



```c
case LEFT_ARROW:


    int cursor = input.e-left_key_pressed_count;

    if (input.w < cursor)
    {
      if (left_key_pressed==0)
      {
        left_key_pressed=1;
      }


      left_key_pressed_count++;
      move_cursor_left();
    }


  break;

case RIGHT_ARROW:

  int cursor1 = input.e-left_key_pressed_count;

  if(input.e>cursor1){
    left_key_pressed_count--;
    move_cursor_right();
  }
  else{
    left_key_pressed=0;
  }
  break;
```

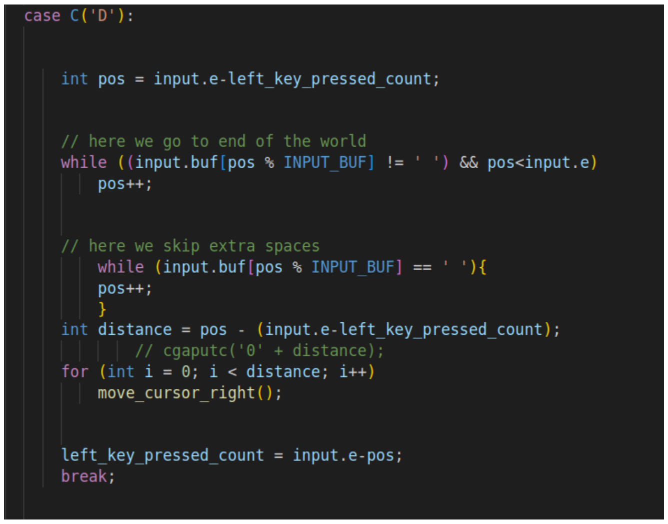
We need to shift the buffer when writing things in the middle or pressing back space (added to default case of console interrupt.)



```c
static void shift_buffer_left(void)
{
  int cursor= input.e-left_key_pressed_count;
  for (int i = cursor - 1; i < input.e; i++)
  {
    input.buf[(i) % INPUT_BUF] = input.buf[(i + 1) % INPUT_BUF]; // Shift elements to left
  }
  input.buf[input.e] = ' ';
}

static void shift_buffer_right(void)
{
   int cursor= input.e-left_key_pressed_count;
  for (int i = input.e; i > cursor; i--)
  {
    input.buf[(i) % INPUT_BUF] = input.buf[(i-1) % INPUT_BUF]; // Shift elements to right
  }
}
```

- **Control D**

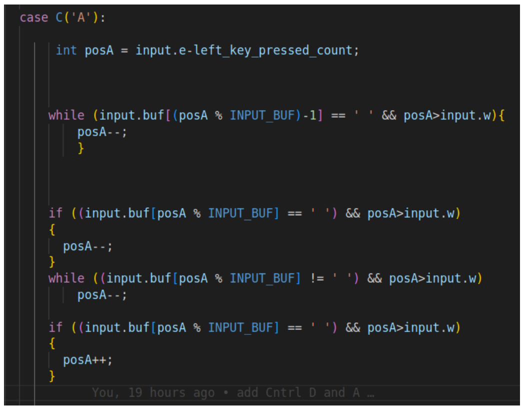
First we go to end of the current word and then we skip extra spaces, and update cursor visual at the end



```c
  case C('D'):

    int pos = input.e-left_key_pressed_count;

    // here we go to end of the world
    while ((input.buf[pos % INPUT_BUF] != ' ') && pos<input.e)
        pos++;

    // here we skip extra spaces
        while (input.buf[pos % INPUT_BUF] == ' '){
        pos++;
        }
    int distance = pos - (input.e-left_key_pressed_count);
            // cgaputc('0' + distance);
    for (int i = 0; i < distance; i++)
        move_cursor_right();

    left_key_pressed_count = input.e-pos;
    break;
```

- **Control A**

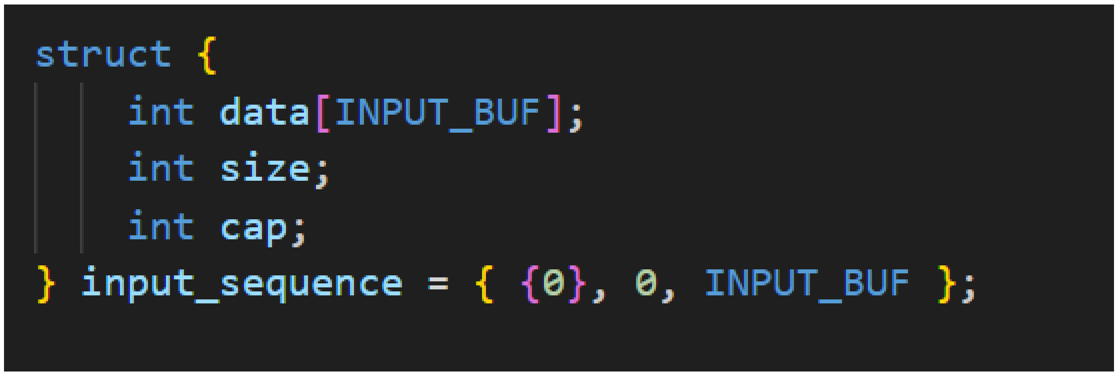
Similar to previous command here we first skip spaces and then skip characters to reach beginning of a word



```c
  case C('A'):

        int posA = input.e-left_key_pressed_count;


       while (input.buf[(posA % INPUT_BUF)-1] == ' ' && posA>input.w){
          posA--;
          }


       if ((input.buf[posA % INPUT_BUF] == ' ') && posA>input.w)
       {
         posA--;
       }
       while ((input.buf[posA % INPUT_BUF] != ' ') && posA>input.w)
          posA--;

       if ((input.buf[posA % INPUT_BUF] == ' ') && posA>input.w)
       {
         posA++;
       }
              You, 19 hours ago • add Cntrl D and A …
```

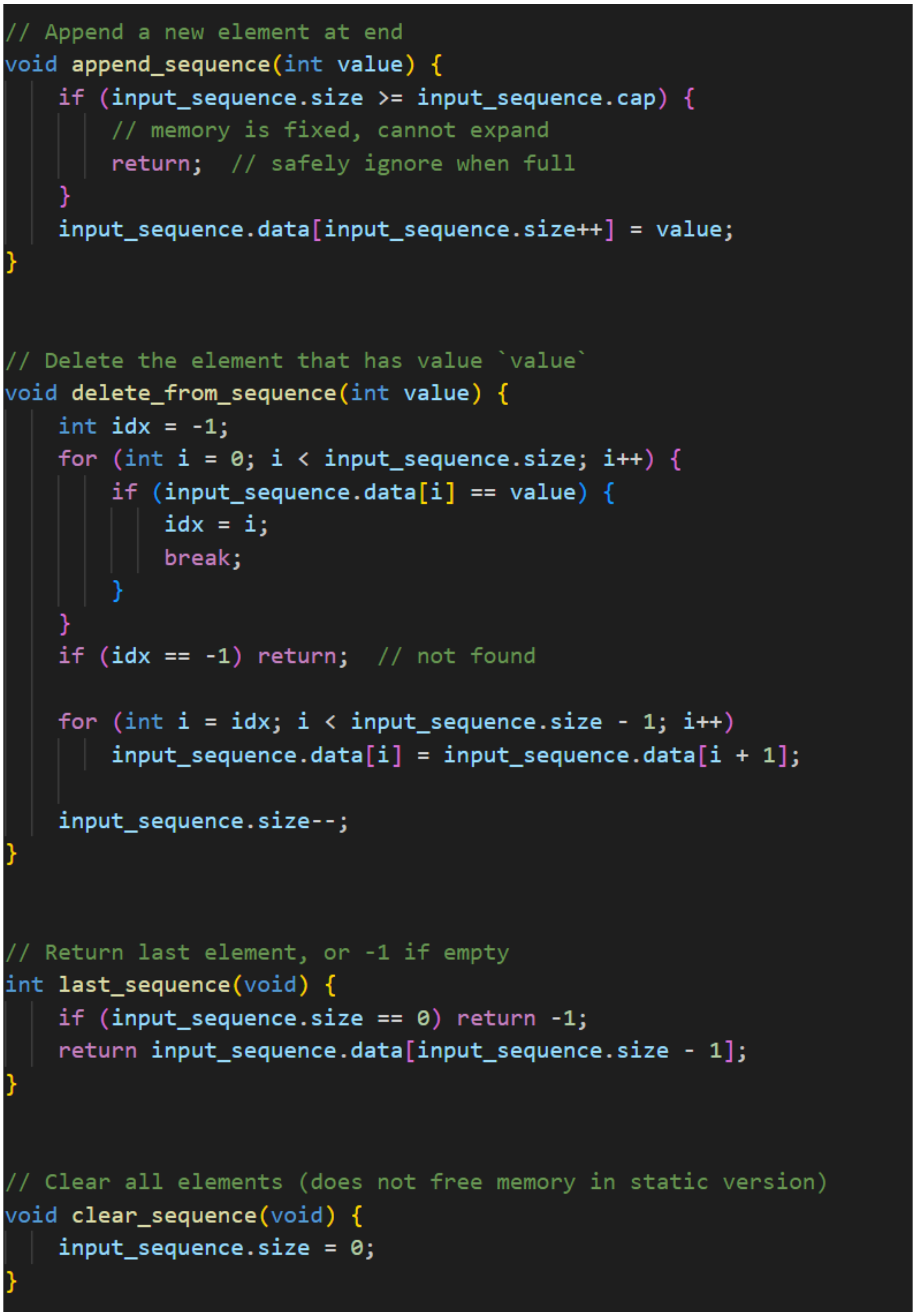
2-Control Z

What we need to do in Ctrl + Z implementation is that we need to make a new data structure that stores the input sequence in the sequence they are being entered . note that their position in input.buf are stored in the structure :



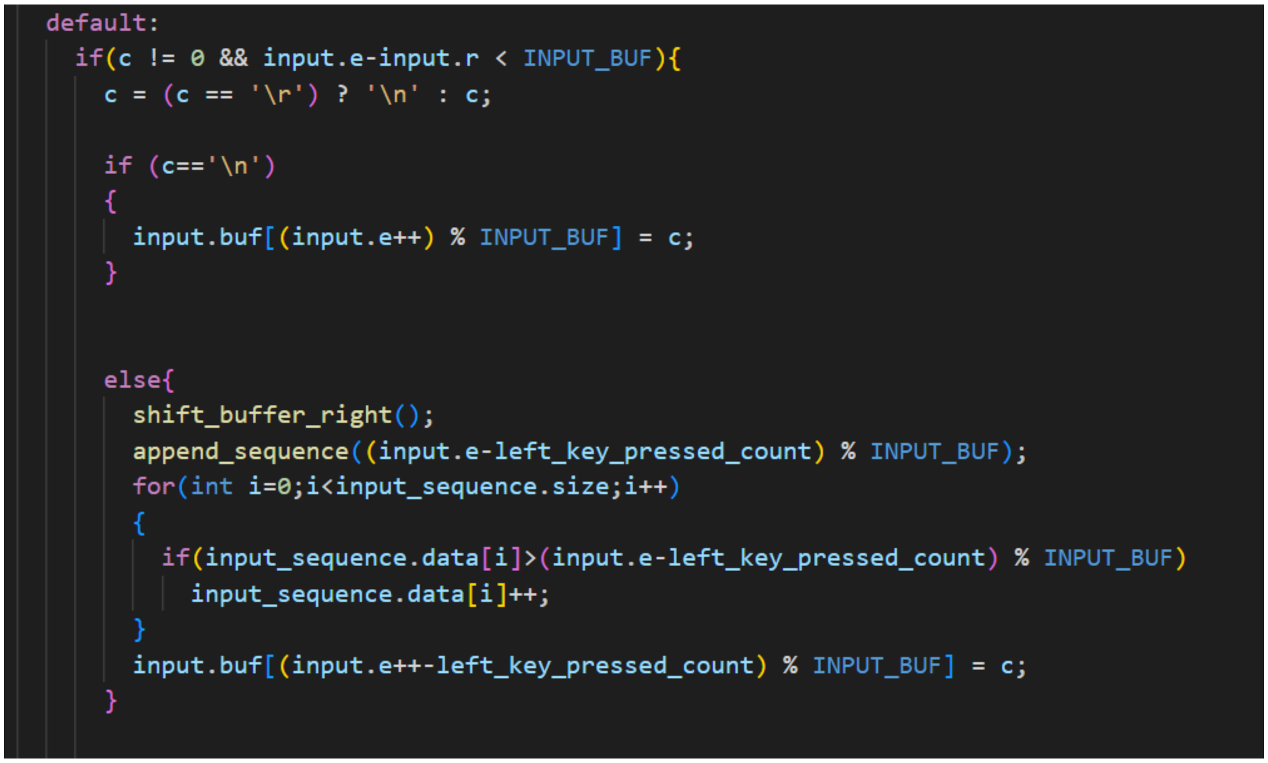
```
struct {
        int data[INPUT_BUF];
        int size;
        int cap;
} input_sequence = { {0}, 0, INPUT_BUF };
```

In this structure we have an array the size of buffer and the size that handles the size of the array so that we can extract the top of the input sequence there are some methods designed for this structure :



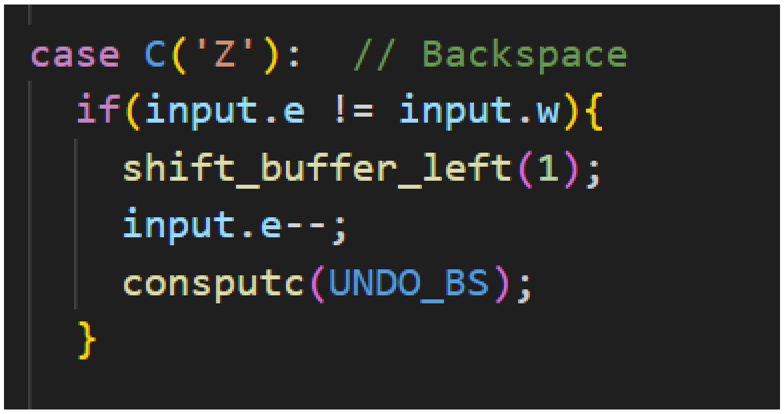
```c
// Append a new element at end
void append_sequence(int value) {
    if (input_sequence.size >= input_sequence.cap) {
        // memory is fixed, cannot expand
        return;  // safely ignore when full
    }
    input_sequence.data[input_sequence.size++] = value;
}


// Delete the element that has value `value`
void delete_from_sequence(int value) {
    int idx = -1;
    for (int i = 0; i < input_sequence.size; i++) {
        if (input_sequence.data[i] == value) {
            idx = i;
            break;
        }
    }
    if (idx == -1) return;  // not found

    for (int i = idx; i < input_sequence.size - 1; i++)
        input_sequence.data[i] = input_sequence.data[i + 1];

    input_sequence.size--;
}


// Return last element, or -1 if empty
int last_sequence(void) {
    if (input_sequence.size == 0) return -1;
    return input_sequence.data[input_sequence.size - 1];
}


// Clear all elements (does not free memory in static version)
void clear_sequence(void) {
    input_sequence.size = 0;
}
```

Now in the consintr function which is handling the interrupts caused by keyboard , in the section where its handling a new character we append the position of the added char in the input_sequence data structure (which is the cursor position in input.buf) and we also need to update the positions in it for when the new char is being added in the middle of our input buffer this pushes the inputs from the right of the cursor to right so we need to increment the positions that are bigger than our cursor pos:



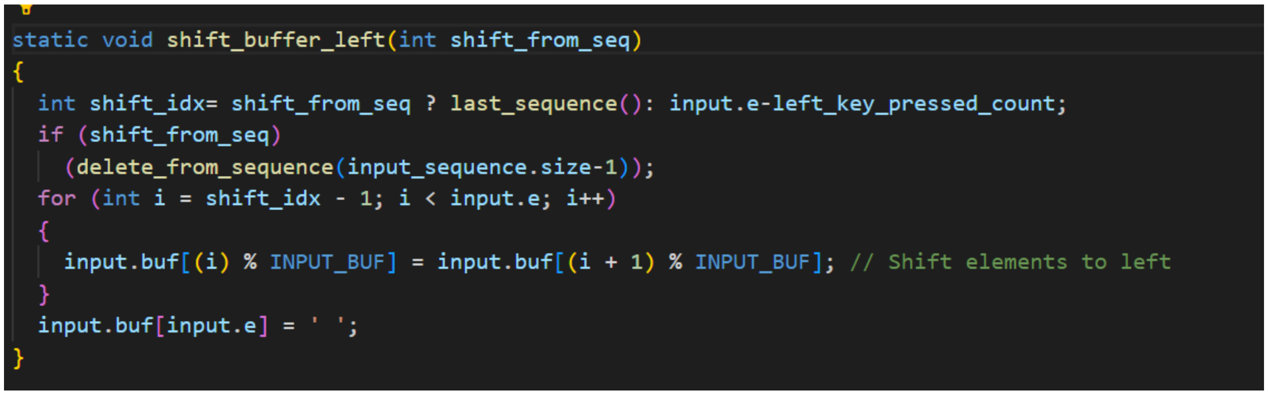
```
default:
  if(c != 0 && input.e-input.r < INPUT_BUF){
    c = (c == '\r') ? '\n' : c;

    if (c=='\n')
    {
      input.buf[(input.e++) % INPUT_BUF] = c;
    }


    else{
      shift_buffer_right();
      append_sequence((input.e-left_key_pressed_count) % INPUT_BUF);
      for(int i=0;i<input_sequence.size;i++)
      {
        if(input_sequence.data[i]>(input.e-left_key_pressed_count) % INPUT_BUF)
          input_sequence.data[i]++;
      }
      input.buf[(input.e++-left_key_pressed_count) % INPUT_BUF] = c;
    }
```

Now we add a new case named C("Z") which should handle the ctrl Z



```
case C('Z'):  // Backspace
  if(input.e != input.w){
    shift_buffer_left(1);
    input.e--;
    consputc(UNDO_BS);
  }
```

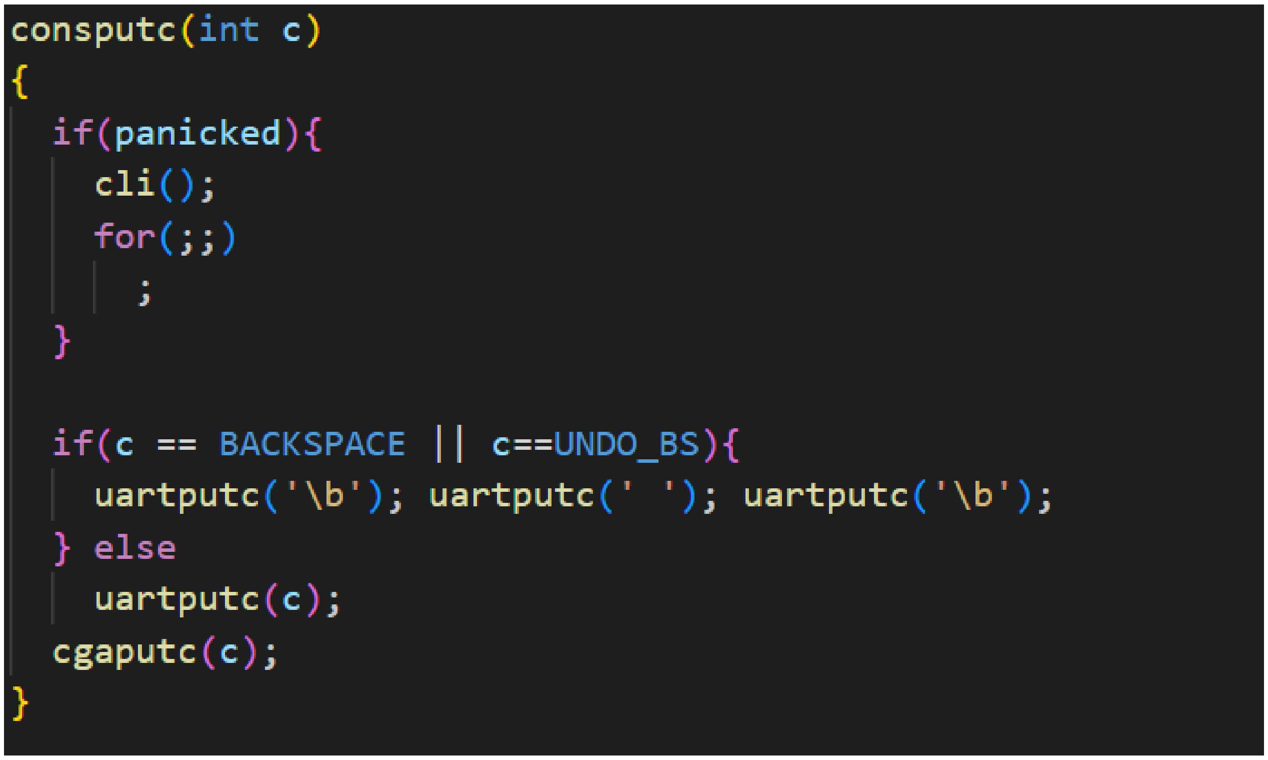
In this we need to shift the buffer to left from the position of the last char entered so we need to make a change to the shift left function :



```
static void shift_buffer_left(int shift_from_seq)
{
  int shift_idx= shift_from_seq ? last_sequence(): input.e-left_key_pressed_count;
  if (shift_from_seq)
    (delete_from_sequence(input_sequence.size-1));
  for (int i = shift_idx - 1; i < input.e; i++)
  {
    input.buf[(i) % INPUT_BUF] = input.buf[(i + 1) % INPUT_BUF]; // Shift elements to left
  }
  input.buf[input.e] = ' ';
}
```

The change we made is that we added an input for it that determines whether the shift should happen from the cursor pos(for general back_space) or the position of the last entered char.

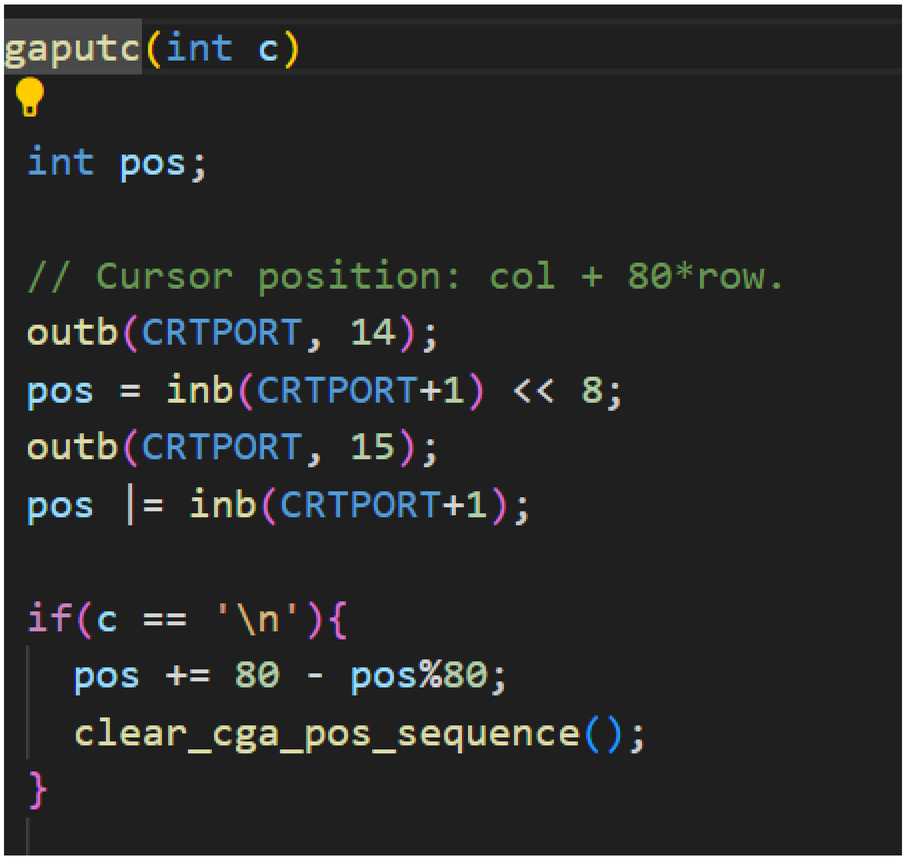
Then I defined a new definition to give the consputc function because we need a different approach for the cgaputc function to delete the last entered char from our terminal display. We called it UNDO_BS.



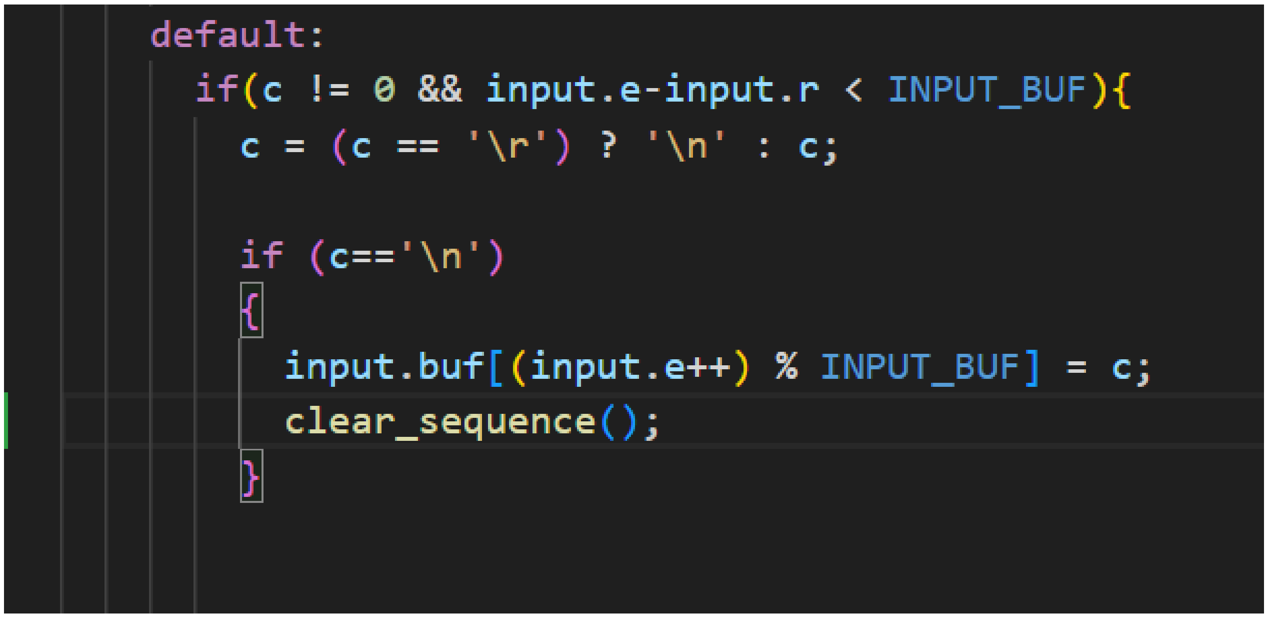
```
consputc(int c)
{
  if(panicked){
    cli();
    for(;;)
      ;
  }

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

In the consputc i handled the uartputc exactly like backspace.
Then i made another structure exactly like the other struct that i used for handling the positions in the display. I handled it exactly like the input buffer .
I stored the position of the new chars in display (using crt array) then handled the update in case of adding new chars in different positions.

NOTE : we need to clear both data structs in case of going to a new line(adding \n) :



```
gaputc(int c)

 int pos;

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

 if(c == '\n'){
   pos += 80 - pos%80;
   clear_cga_pos_sequence();
 }
```

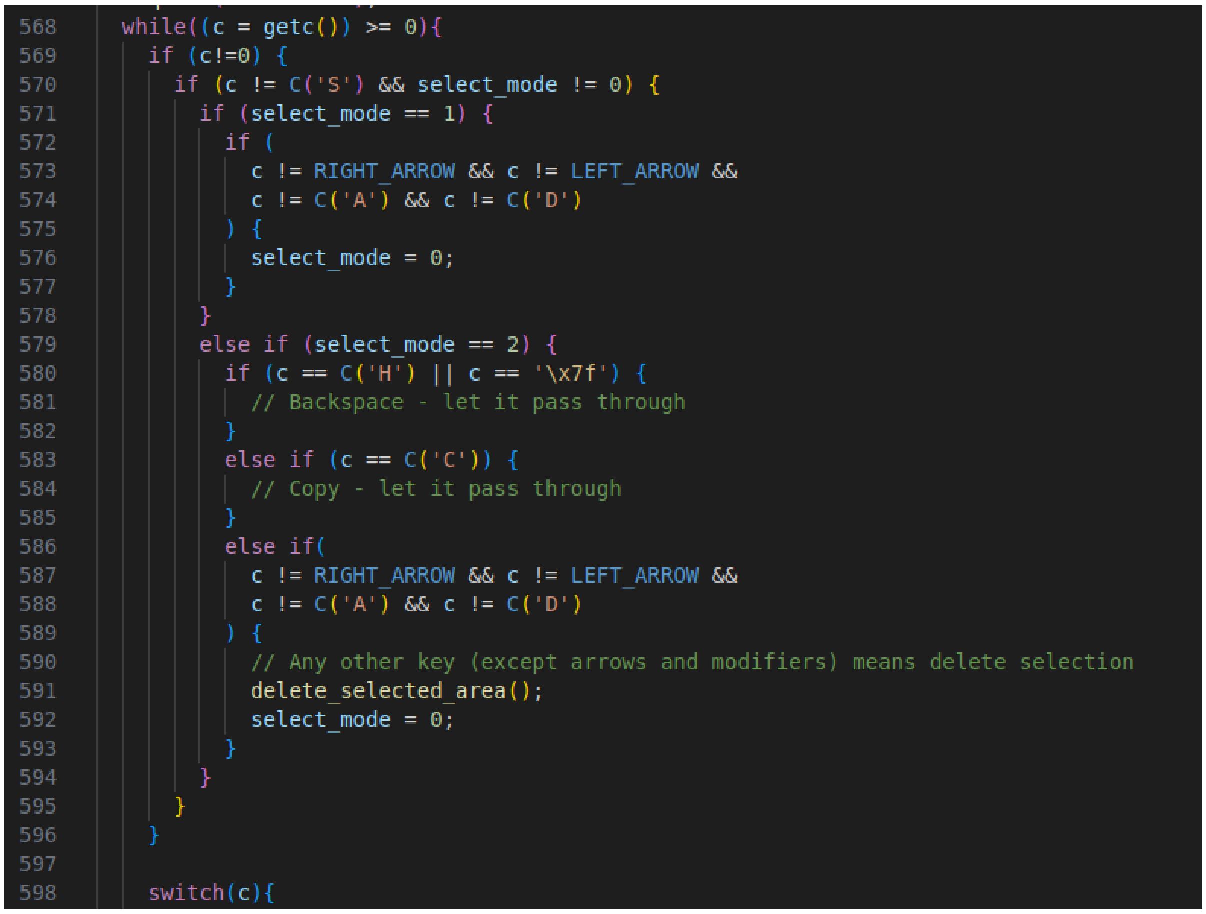


```
      default:
        if(c != 0 && input.e-input.r < INPUT_BUF){
          c = (c == '\r') ? '\n' : c;

          if (c=='\n')
          {
            input.buf[(input.e++) % INPUT_BUF] = c;
            clear_sequence();
          }
```

- **Select-copy-paste**

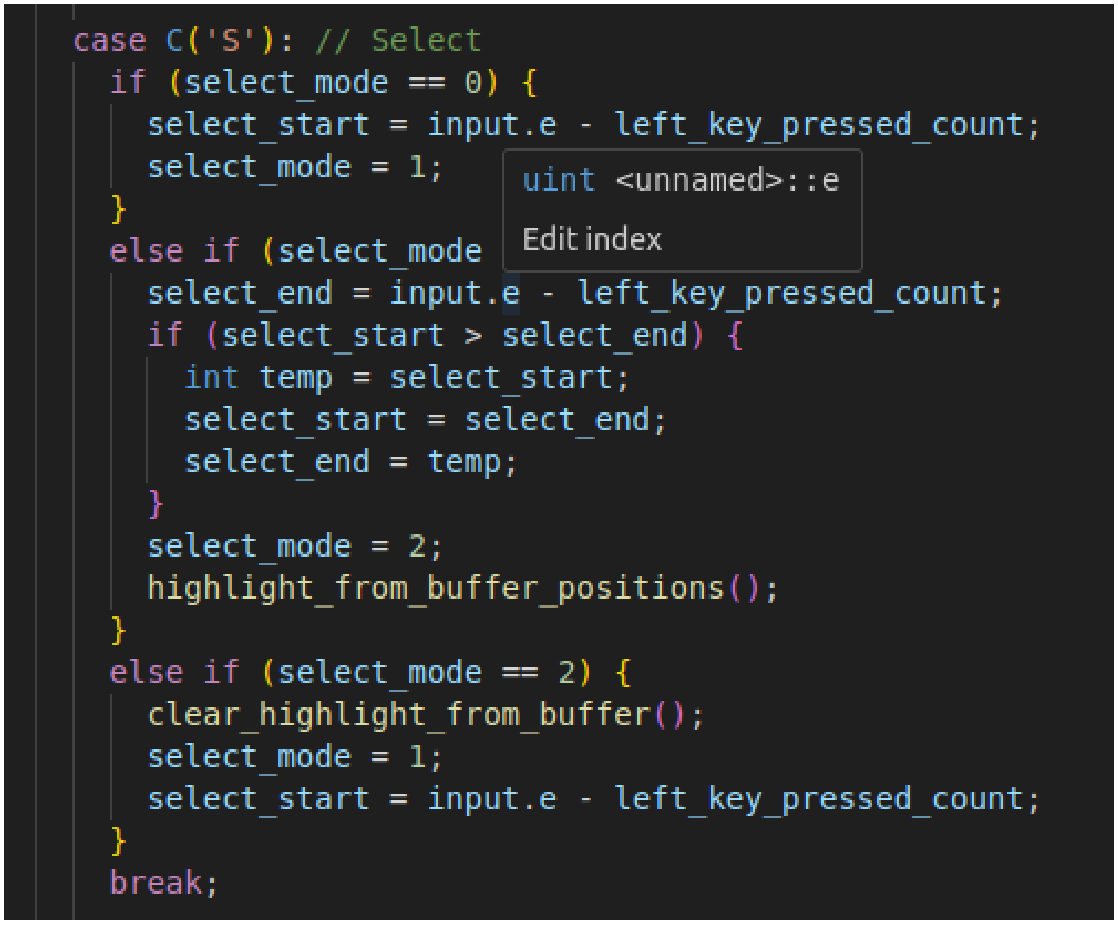
0. Select / Control S

The main idea for select is to count the number of times ctrl S has been pressed without getting canceled, and that is why we use 3 states to keep the select mode, first is no selects has been pressed, second is one ctrl S has been pressed and last state is for when we are done selecting a section.
The point is in order to reset the select we need to check buttons that reset it and it is hard and could be changed, so instead before entering switch case we use an if statement to reset select mode for when we are not using arrows or ctrl D and A. This way we need not modify other parts of the switch like the default.
Here the code showing how select mode is updated and reseted:



```
  while((c = getc()) >= 0){
    if (c!=0) {
      if (c != C('S') && select_mode != 0) {
        if (select_mode == 1) {
          if (
            c != RIGHT_ARROW && c != LEFT_ARROW &&
            c != C('A') && c != C('D')
          ) {
            select_mode = 0;
          }
        }
        else if (select_mode == 2) {
          if (c == C('H') || c == '\x7f') {
            // Backspace - let it pass through
          }
          else if (c == C('C')) {
            // Copy - let it pass through
          }
          else if(
            c != RIGHT_ARROW && c != LEFT_ARROW &&
            c != C('A') && c != C('D')
          ) {
            // Any other key (except arrows and modifiers) means delete selection
            delete_selected_area();
            select_mode = 0;
          }
        }
      }
    }

    switch(c){
```

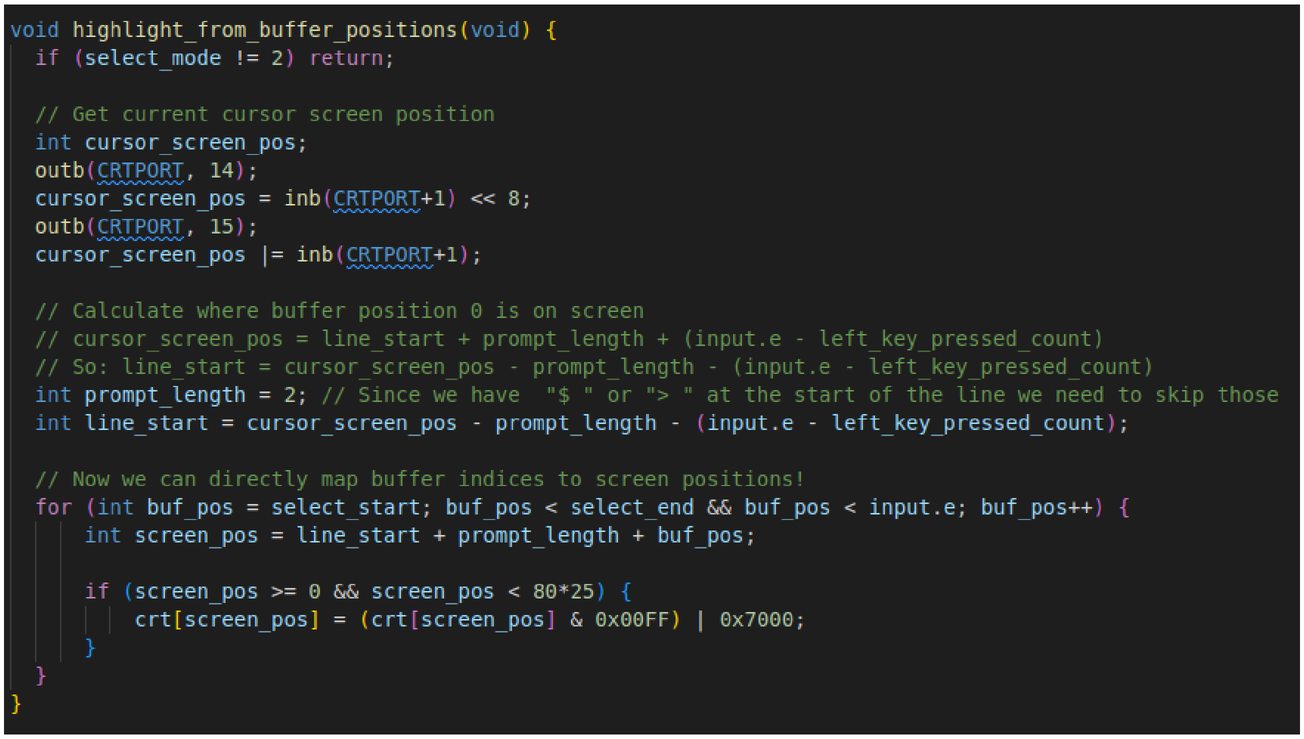
As it can be seen some cases like backspace and ctrl C are later handled and also select mode does not reset after moving the cursor.
And in select we keep two pointer to keep start and end and also swap them if the first select is more to the right than the end of select:



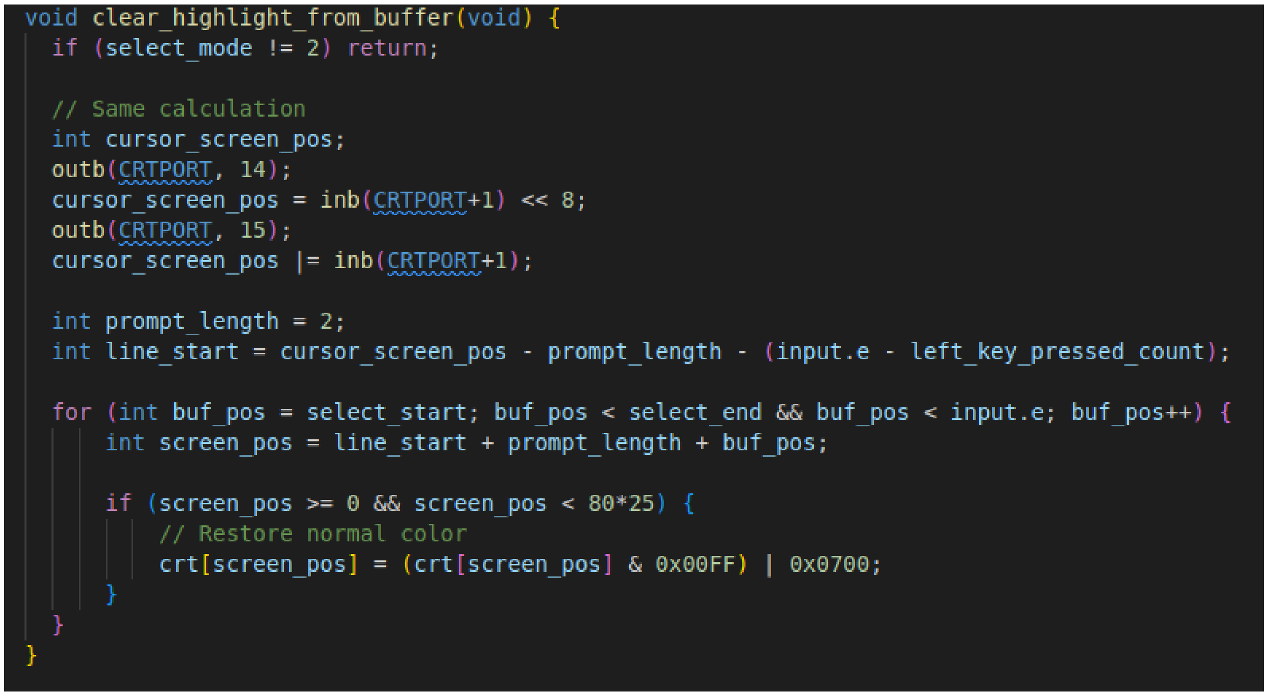
```
case C('S'): // Select
  if (select_mode == 0) {
    select_start = input.e - left_key_pressed_count;
    select_mode = 1;
  }
  else if (select_mode
    select_end = input.e - left_key_pressed_count;
    if (select_start > select_end) {
      int temp = select_start;
      select_start = select_end;
      select_end = temp;
    }
    select_mode = 2;
    highlight_from_buffer_positions();
  }
  else if (select_mode == 2) {
    clear_highlight_from_buffer();
    select_mode = 1;
    select_start = input.e - left_key_pressed_count;
  }
  break;
```

But there's one big point left, and it is to highlight this section, that's why we write two more functions, one to highlight from start to end and one to remove the highlight from this exact location. But since this had to be done visually we need to first get the actual location we need to change color and then change it:

*Using x7000 make the background grey and the text itself black and x0700 which is the default makes the text grey and the background black.



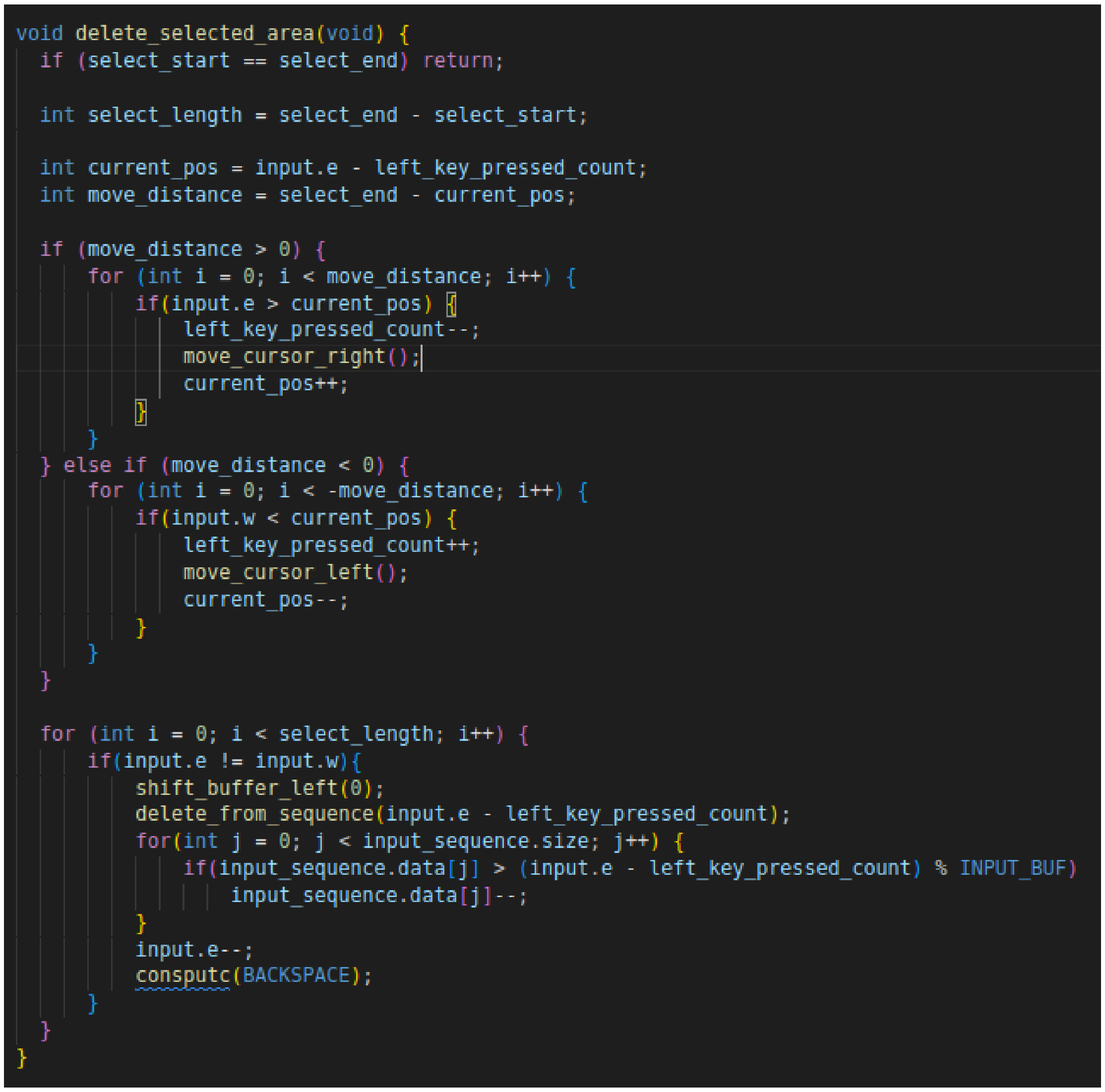
```c
void highlight_from_buffer_positions(void) {
  if (select_mode != 2) return;

  // Get current cursor screen position
  int cursor_screen_pos;
  outb(CRTPORT, 14);
  cursor_screen_pos = inb(CRTPORT+1) << 8;
  outb(CRTPORT, 15);
  cursor_screen_pos |= inb(CRTPORT+1);

  // Calculate where buffer position 0 is on screen
  // cursor_screen_pos = line_start + prompt_length + (input.e - left_key_pressed_count)
  // So: line_start = cursor_screen_pos - prompt_length - (input.e - left_key_pressed_count)
  int prompt_length = 2; // Since we have  "$ " or "> " at the start of the line we need to skip those
  int line_start = cursor_screen_pos - prompt_length - (input.e - left_key_pressed_count);

  // Now we can directly map buffer indices to screen positions!
  for (int buf_pos = select_start; buf_pos < select_end && buf_pos < input.e; buf_pos++) {
    int screen_pos = line_start + prompt_length + buf_pos;

    if (screen_pos >= 0 && screen_pos < 80*25) {
      crt[screen_pos] = (crt[screen_pos] & 0x00FF) | 0x7000;
    }
  }
}
```



```c
void clear_highlight_from_buffer(void) {
  if (select_mode != 2) return;

  // Same calculation
  int cursor_screen_pos;
  outb(CRTPORT, 14);
  cursor_screen_pos = inb(CRTPORT+1) << 8;
  outb(CRTPORT, 15);
  cursor_screen_pos |= inb(CRTPORT+1);

  int prompt_length = 2;
  int line_start = cursor_screen_pos - prompt_length - (input.e - left_key_pressed_count);

  for (int buf_pos = select_start; buf_pos < select_end && buf_pos < input.e; buf_pos++) {
    int screen_pos = line_start + prompt_length + buf_pos;

    if (screen_pos >= 0 && screen_pos < 80*25) {
      // Restore normal color
      crt[screen_pos] = (crt[screen_pos] & 0x00FF) | 0x0700;
    }
  }
}
```

1. Delete / Backspace

For this part we are going to write a function which works like this:
It sets the cursor's location to the end of the selection then using a for acts like a number of backspaces had been pressed in this spot, it is pretty simple and has two main parts, moving the cursor, and them using a for loop to delete the section.

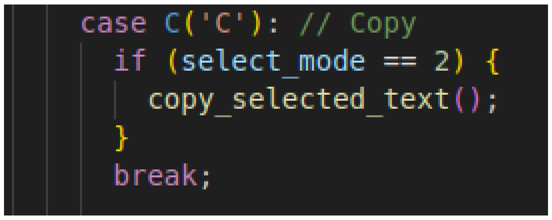


```c
void delete_selected_area(void) {
  if (select_start == select_end) return;

  int select_length = select_end - select_start;

  int current_pos = input.e - left_key_pressed_count;
  int move_distance = select_end - current_pos;

  if (move_distance > 0) {
    for (int i = 0; i < move_distance; i++) {
      if(input.e > current_pos) {
        left_key_pressed_count--;
        move_cursor_right();
        current_pos++;
      }
    }
  } else if (move_distance < 0) {
    for (int i = 0; i < -move_distance; i++) {
      if(input.w < current_pos) {
        left_key_pressed_count++;
        move_cursor_left();
        current_pos--;
      }
    }
  }

  for (int i = 0; i < select_length; i++) {
    if(input.e != input.w){
      shift_buffer_left(0);
      delete_from_sequence(input.e - left_key_pressed_count);
      for(int j = 0; j < input_sequence.size; j++) {
        if(input_sequence.data[j] > (input.e - left_key_pressed_count) % INPUT_BUF)
          input_sequence.data[j]--;
      }
      input.e--;
      consputc(BACKSPACE);
    }
  }
}
```

And we call it in many cases, some said further in and one in backspace when select mode is 2.

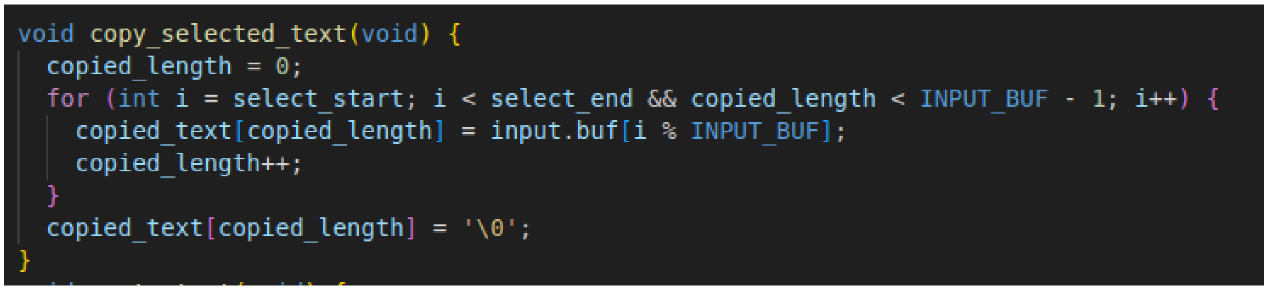
2. Control C / Copy

This part is very simple, it uses a for to copy from start to end in another array and also keeps its length.



```c
    case C('C'): // Copy
      if (select_mode == 2) {
        copy_selected_text();
      }
      break;
```

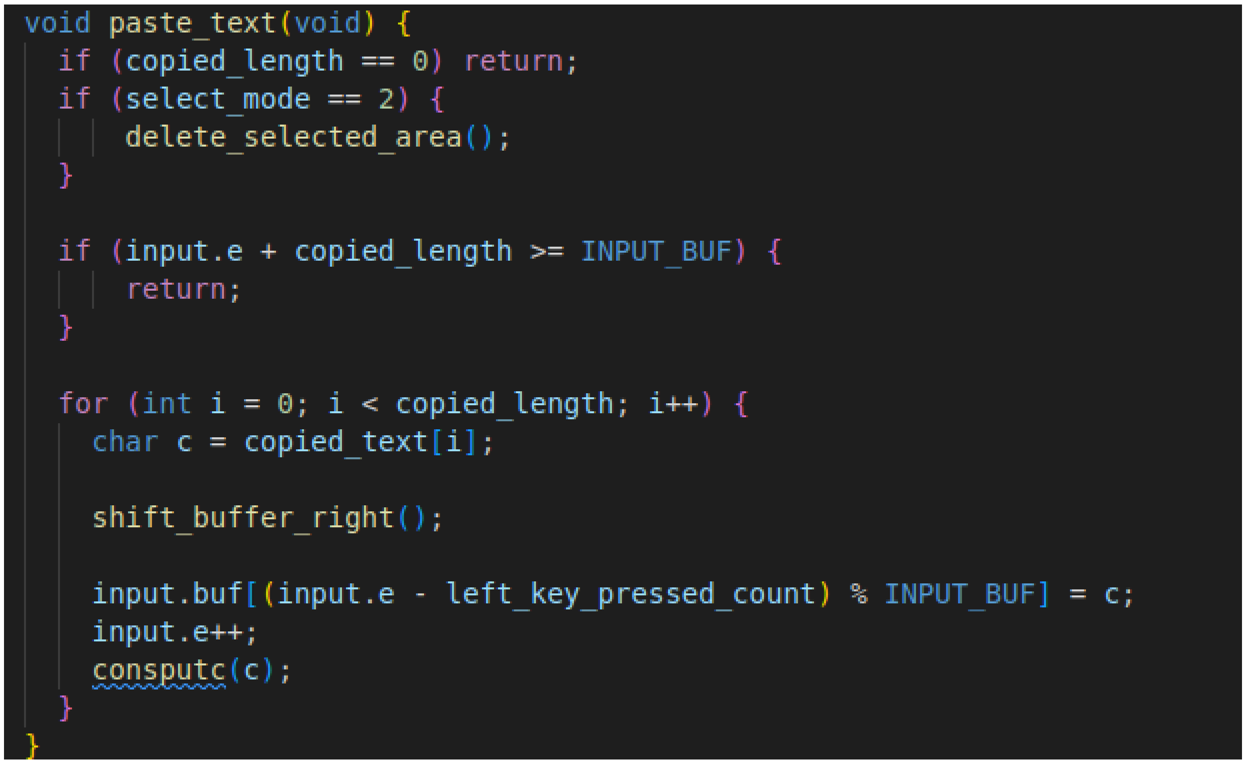
Also add a ‘\0’ to get the end of it although it is not needed.



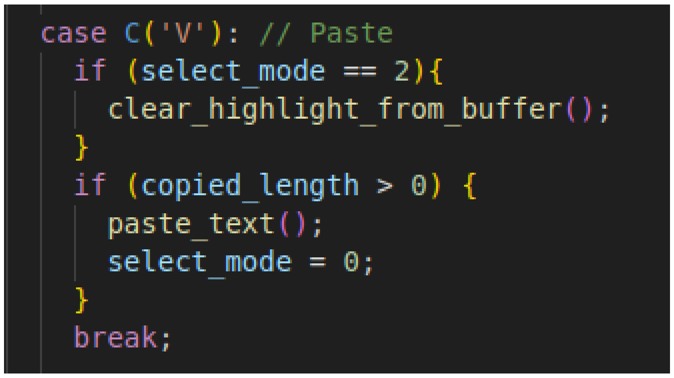
```c
void copy_selected_text(void) {
  copied_length = 0;
  for (int i = select_start; i < select_end && copied_length < INPUT_BUF - 1; i++) {
    copied_text[copied_length] = input.buf[i % INPUT_BUF];
    copied_length++;
  }
  copied_text[copied_length] = '\0';
}
```

3. **Control V / Paste**

If the select mod is not 2 meaning we need to paste in the exact location we just paste like deleting and use a for to add chars from the copied text to the buffer, otherwise we need to delete the selection and then paste there, and since we have done the delete in a way that it correctly move the cursor to the location we want it to be in we can simply can delete and then paste.



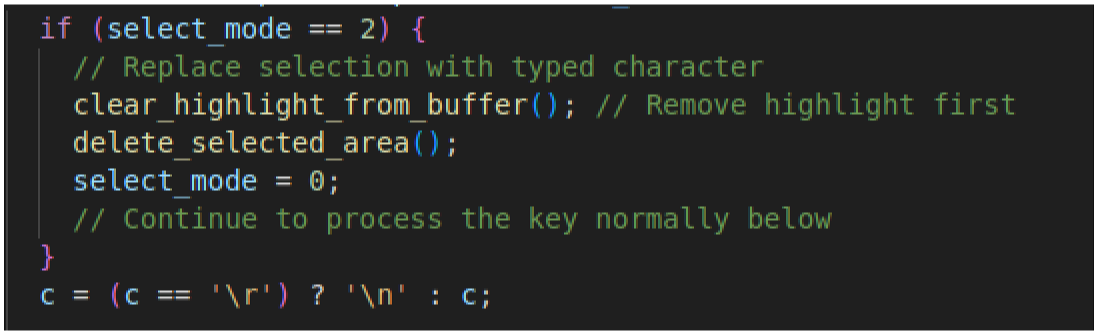
```c
void paste_text(void) {
  if (copied_length == 0) return;
  if (select_mode == 2) {
    delete_selected_area();
  }

  if (input.e + copied_length >= INPUT_BUF) {
    return;
  }

  for (int i = 0; i < copied_length; i++) {
    char c = copied_text[i];

    shift_buffer_right();

    input.buf[(input.e - left_key_pressed_count) % INPUT_BUF] = c;
    input.e++;
    consputc(c);
  }
}
```



```c
  case C('V'): // Paste
    if (select_mode == 2){
      clear_highlight_from_buffer();
    }
    if (copied_length > 0) {
      paste_text();
      select_mode = 0;
    }
    break;
```

4. Replace char with selected section

Similar to paste but easier, although this is handled in the default section, the way it works is it simply deletes the selected area and since the cursor is now in the correct spot it then replaces the char with it. So it is handled as followed:



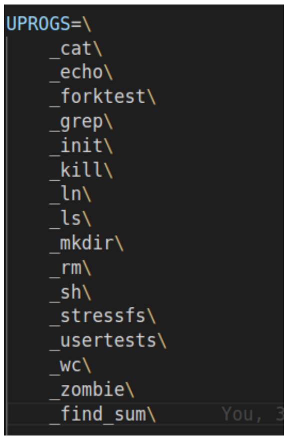
```c
if (select_mode == 2) {
  // Replace selection with typed character
  clear_highlight_from_buffer(); // Remove highlight first
  delete_selected_area();
  select_mode = 0;
  // Continue to process the key normally below
}
c = (c == '\r') ? '\n' : c;
```

As you can see we also update the highlighting when needed and use functions to do a specific job for us.

*** A part in select was changed close to deadline, now in mode 2 after using arrow and moving cursor and ctrl Z and ctrl S it gets reset***

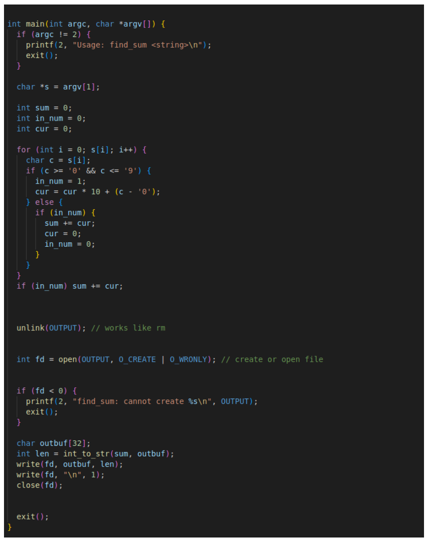
User level program

First we add find_sum to user programs in makefile



```
UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_usertests\
	_wc\
	_zombie\
	_find_sum\        You, 3
```

Then we make the actual code like any other c program



```c
int main(int argc, char *argv[]) {
  if (argc != 2) {
    printf(2, "Usage: find_sum <string>\n");
    exit();
  }

  char *s = argv[1];

  int sum = 0;
  int in_num = 0;
  int cur = 0;

  for (int i = 0; s[i]; i++) {
    char c = s[i];
    if (c >= '0' && c <= '9') {
      in_num = 1;
      cur = cur * 10 + (c - '0');
    } else {
      if (in_num) {
        sum += cur;
        cur = 0;
        in_num = 0;
      }
    }
  }
  if (in_num) sum += cur;



  unlink(OUTPUT); // works like rm


  int fd = open(OUTPUT, O_CREATE | O_WRONLY); // create or open file


  if (fd < 0) {
    printf(2, "find_sum: cannot create %s\n", OUTPUT);
    exit();
  }

  char outbuf[32];
  int len = int_to_str(sum, outbuf);
  write(fd, outbuf, len);
  write(fd, "\n", 1);
  close(fd);


  exit();
}
```

10):

UPROGS:

This variable holds all user-level exe that are in xv6's file system.

Each item is a binary file that will appear in / inside xv6 (like /ls, /cat, /echo, etc.).meaning when you boot xv6 these programs are in root and can be run.

ULIB:

This one is all user's library files (like ulib.c, printf.c, umalloc.c, etc.) that they can use.you can include these in all user programs without re implementing them

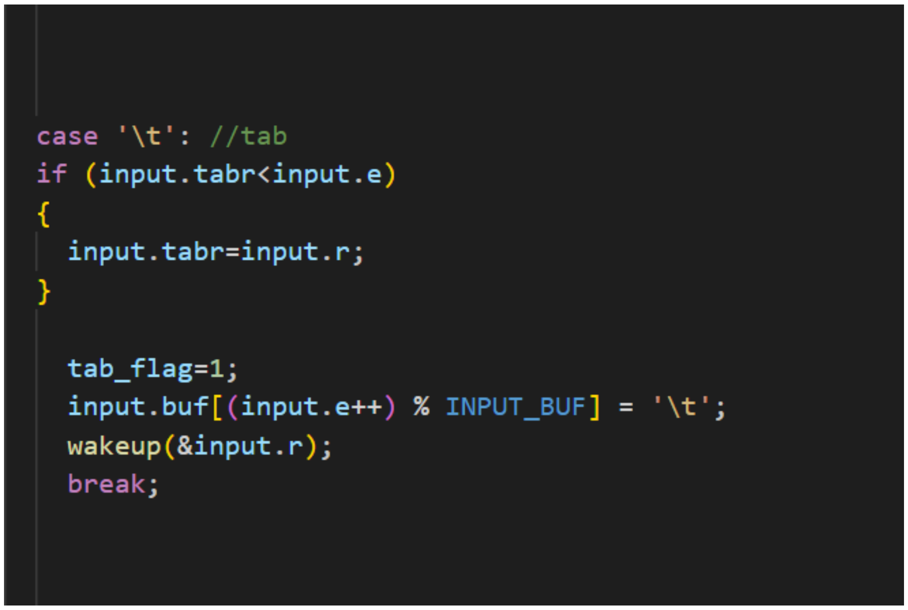
11):

In xv6, the cd command does not exist as a separate program because it needs to change the shell's current working directory . If cd were a normal executable, it would run in a child process, and changing the directory there would not affect the shell itself. Cd is implemented as a built-in command inside the shell, using the a system call to update the shell's cwd. Other commands like ls and cat do not need to modify the shell state, so they can run as normal programs in child processes.(they are only reading)

Autocompletion

The core approach is to handle the connections between shell and console . we have to send the incomplete command to the shell and we find its matches and rewrite the correct match if there is only one match and if there are multiple matches we print the matches and then rewrite the incomplete command and proceed with the rest.

The first step in its implementation is we need to add a case for Tab in consoleintr:



```
case '\t': //tab
if (input.tabr<input.e)
{
  input.tabr=input.r;
}

  tab_flag=1;
  input.buf[(input.e++) % INPUT_BUF] = '\t';
  wakeup(&input.r);
  break;
```

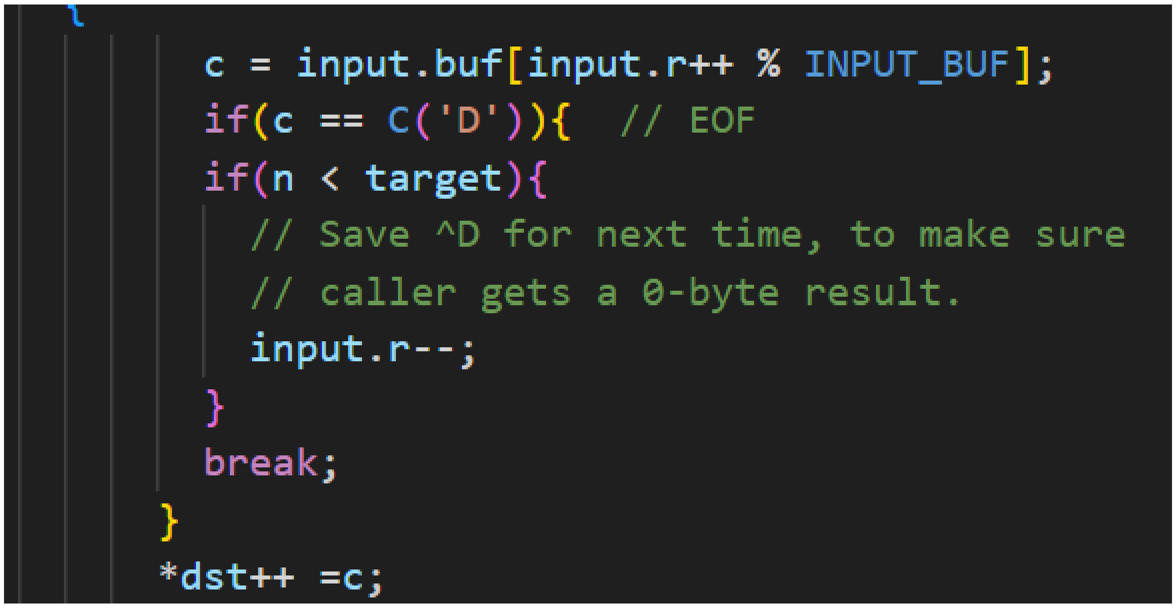
In this case we add a '\t' to the input buffer and wake the shell up(so that it would send the commands that have been entered before the tab.

In order to do that we need to add a input.tabr that we'll explain in the next page.

And also we need to have a flag that shows whether the tab has been pushed or not because we need to know that in the console read which read pointer we need to increment.

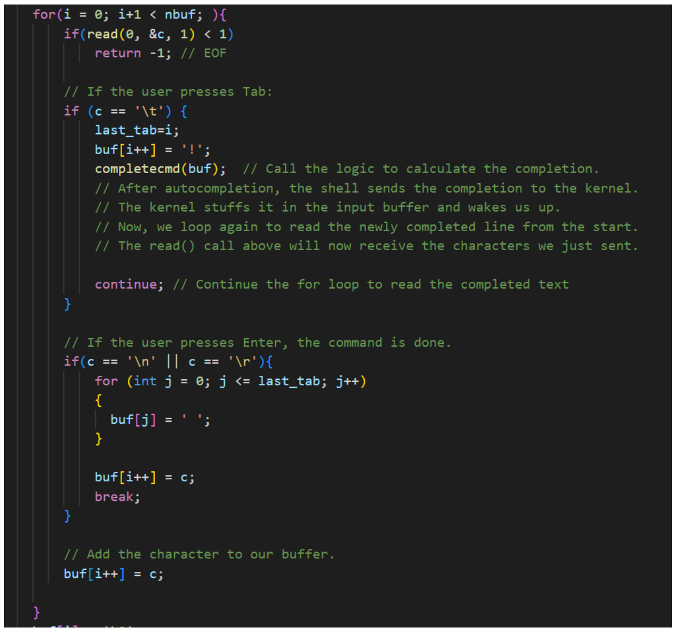
We also add a if at the beginning that if the input.tabr < input.e then set the input.tabr to input.r this is because if we already had pushed tab the input.tab.r has reached to the end and the tab actions have been handled and we shouldn't update input.tabr.

First we add a input.tabr in the input struct. This would be a copy of the input r when we press tab. The reason we need to have a copy of r is that the structure of sending to shell is like this:



```c
      c = input.buf[input.r++ % INPUT_BUF];
      if(c == C('D')){  // EOF
      if(n < target){
        // Save ^D for next time, to make sure
        // caller gets a 0-byte result.
        input.r--;
      }
      break;
    }
    *dst++ =c;
```

Lets say we have pushed \n and we have woken up the consoleread this piece of of code is moving the input.r towards the end and sending them to the shells buffer . then in the shell We have a getcmd function:

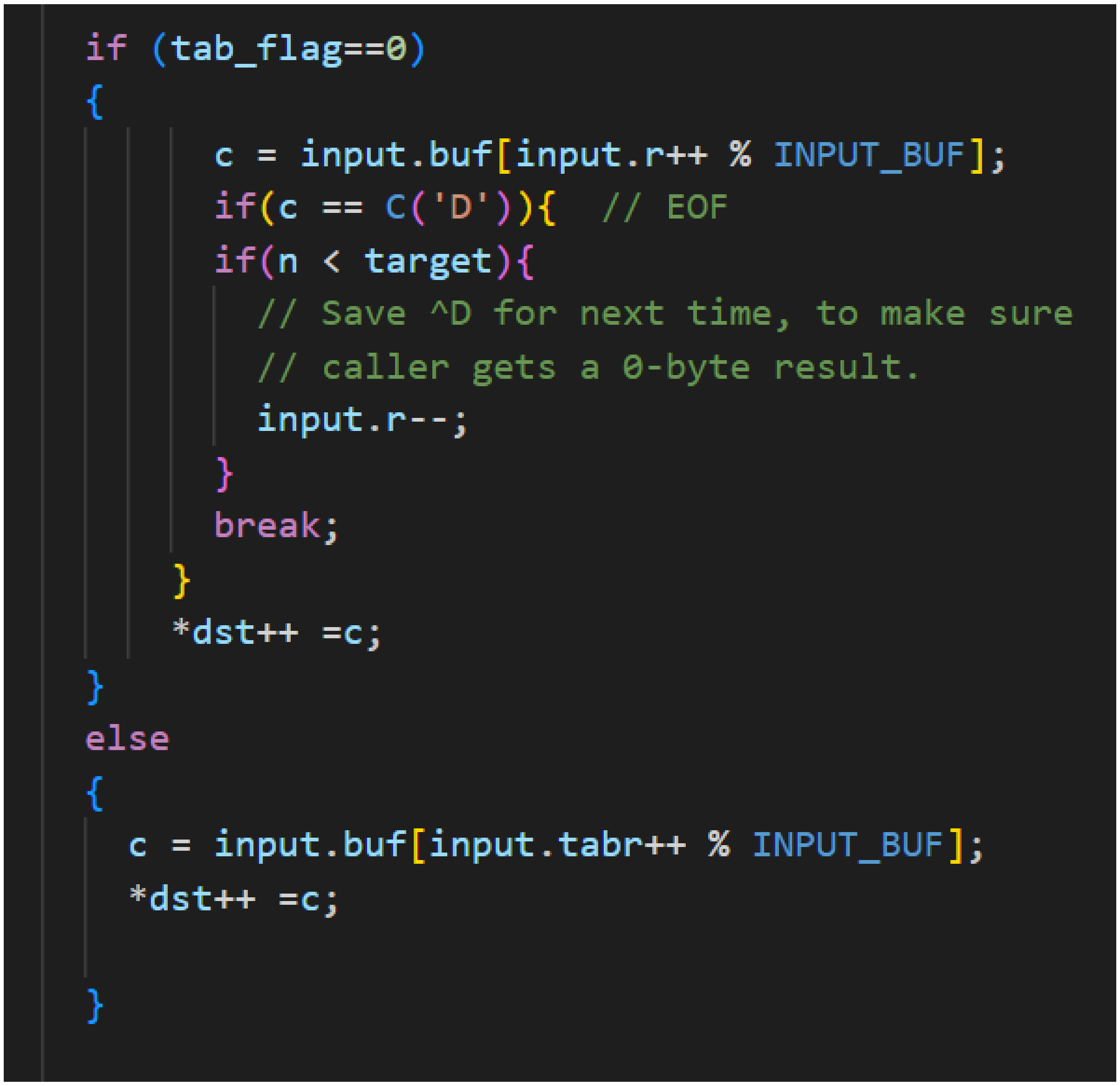


```c
  for(i = 0; i+1 < nbuf; ){
    if(read(0, &c, 1) < 1)
      return -1; // EOF

    // If the user presses Tab:
    if (c == '\t') {
      last_tab=i;
      buf[i++] = '!';
      completecmd(buf);  // Call the logic to calculate the completion.
      // After autocompletion, the shell sends the completion to the kernel.
      // The kernel stuffs it in the input buffer and wakes us up.
      // Now, we loop again to read the newly completed line from the start.
      // The read() call above will now receive the characters we just sent.

      continue; // Continue the for loop to read the completed text
    }

    // If the user presses Enter, the command is done.
    if(c == '\n' || c == '\r'){
      for (int j = 0; j <= last_tab; j++)
      {
        buf[j] = ' ';
      }

      buf[i++] = c;
      break;
    }

    // Add the character to our buffer.
    buf[i++] = c;

  }
```

It reads one char from console read and gets the char from the input.buf then adds it into the shells buf and breaks the loop if ‘\n’ is entered to execute it.

The approach for tab is similar to this but we shouldn't execute the incomplete command. Instead we should find the matches for the complete command and send it back to the console so that the user could write the rest of the line.

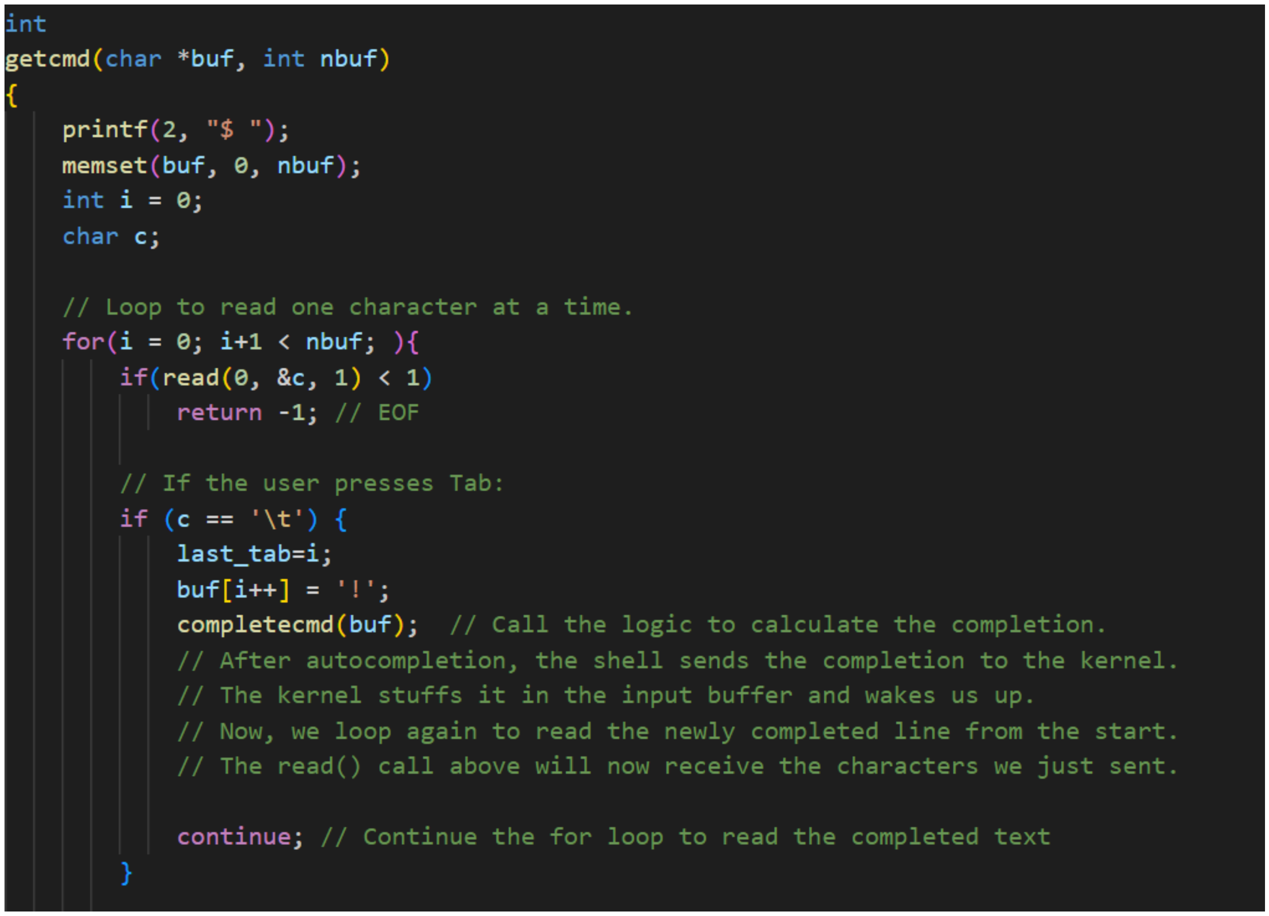
That's why we need the input.tabr we have to keep the unexecuted pointer of our input buffer (which is input.r and in the same time move it towards the end to send the incomplete chars to shell) so now we add a condition in console read that if we have pushed the tab(tab flag ==1) then we move the input tabr:



```
if (tab_flag==0)
{
      c = input.buf[input.r++ % INPUT_BUF];
      if(c == C('D')){  // EOF
      if(n < target){
        // Save ^D for next time, to make sure
        // caller gets a 0-byte result.
        input.r--;
      }
      break;
    }
    *dst++ =c;
}
else
{
  c = input.buf[input.tabr++ % INPUT_BUF];
  *dst++ =c;

}
```

This way if we press tab the else section will be performed and the incomplete command gets sent to the shells buffer.

Now in the shell we have a getcmd function in it that we



```
int
getcmd(char *buf, int nbuf)
{
    printf(2, "$ ");
    memset(buf, 0, nbuf);
    int i = 0;
    char c;

    // Loop to read one character at a time.
    for(i = 0; i+1 < nbuf; ){
        if(read(0, &c, 1) < 1)
            return -1; // EOF

        // If the user presses Tab:
        if (c == '\t') {
            last_tab=i;
            buf[i++] = '!';
            completecmd(buf);  // Call the logic to calculate the completion.
            // After autocompletion, the shell sends the completion to the kernel.
            // The kernel stuffs it in the input buffer and wakes us up.
            // Now, we loop again to read the newly completed line from the start.
            // The read() call above will now receive the characters we just sent.

            continue; // Continue the for loop to read the completed text
        }
```

The read function goes to console read and receives the chars that we are sending to the shell char by char .

We have added an if condition that will be handling the completecmd function in which we add an '!' (this is a pointer , we add it to keep trace of the tabs i will explain it later and also we store the last position of the tab in last tab) then we proceed to completecmd.

These two `static` variables are crucial. Being `static`, they retain their values between function calls.

- `last_prefix`: Stores the partial word from the *last* time `Tab` was pressed.
- `tab_count`: Counts how many times `Tab` has been pressed *consecutively* for the *same* partial word.

This is how the shell distinguishes between a single tab press (which might auto-complete) and a double tab press (which should show all options).

`getprefix` iterates through the buffer and copies every character into the `prefix` variable until it hits that `!` marker.

It then null-terminates `prefix`.

Example: If the user types `l`, `s`, and then `c`, `a`, `Tab`, the `buf` in `getcmd` will become `"ls ca!"`. The `getprefix` function will extract `"ls ca"` into the `prefix` variable.

Then **If the current prefix is different from the last one:** It means the user is trying to complete a *new* word. The code resets `tab_count` to `1` and updates `last_prefix` to this new word.

And If the current prefix is the same as the last one: It means the user pressed `Tab` again for the same word. The code simply increments `tab_count`.

First, it looks at our small list of `builtins`.

1. It loops through the `builtins` array (`"cd"`).
2. `strncmp(prefix, builtins[i], strlen(prefix))` compares the beginning of the built-in command with the user's prefix.
3. **If they match** (e.g., prefix is `"c"` and built-in is `"cd"`), it copies the full built-in command name into the `matches` array and increments `match_count`.

Now we need to search through the rest of the commands:

1. `open(".", 0)` opens the current directory for reading.
2. It then enters a `while` loop, using `read` to get directory entries (`struct dirent`) one by one. A directory in xv6 is just a file containing a list of these structures.
3. `if (de.inum == 0) continue;` skips unused or deleted entries.
4. `strncmp(prefix, de.name, strlen(prefix))` compares the user's prefix with the beginning of the filename (`de.name`).
5. **If they match** (e.g., prefix is `"l"` and filename is `"ls"`), it copies the full filename into the `matches` array and increments `match_count`.
6. `close(fd)` releases the directory file handle when done.

At the end of this step, `matches` contains a complete list of all built-ins and files that start with the user's prefix.

Now we have to decide the actions based on the matches count:

Case 1: **`match_count == 0`**

No commands or files match the prefix. The function does nothing and returns.

Case 2: **`match_count == 1`**

Exactly one match was found (e.g., user typed `mk` and only `mkdir` exists).

This is the auto-completion case.

`printf(2,"\t%s\t", matches[0]);` prints the single match.

We have added this part in this format so that we can detect it in the console write .

We'll explain the console write later.

Case 3: **`match_count > 1`** **and** **`tab_count == 1`**

There are multiple possibilities (e.g., user typed `g` and both `grep` and `grind` exist), and this is the *first* time they hit tab.

Standard shell behavior is to do nothing, waiting for a second tab press to confirm the user wants to see the list. Your code correctly implements this by having an empty block.

Case 4: **`match_count > 1`** **and** **`tab_count > 1`**

There are multiple possibilities, and the user has pressed `Tab` a second (or third, etc.) time.

The code prints `\nMatches:\n` and then loops through the `matches` array, printing all the possibilities to the screen for the user to see. It also re-prints the prompt and the partial word the user had typed.

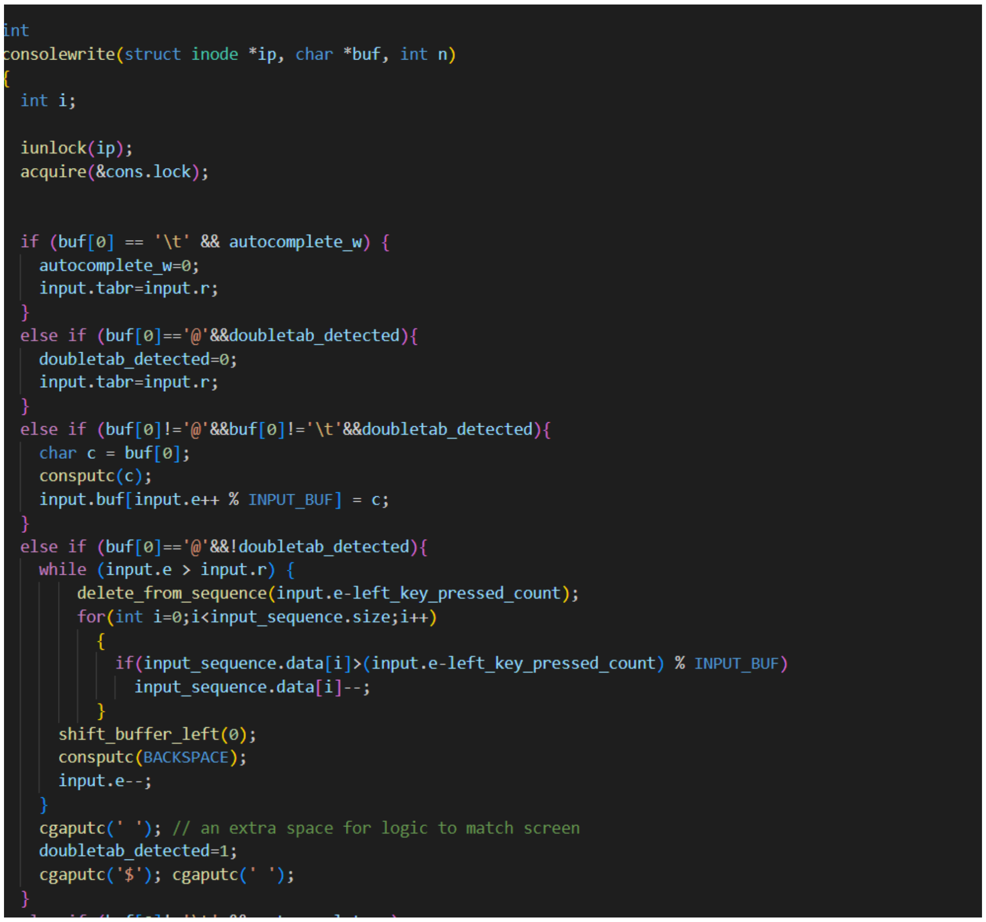
Then we print `printf(2,"@%s@",buf);`

This is again a format to detect it in the console write.

We send the buf to the console because it contains the incomplete command .

After finishing the completecmd function we get back in the getcmd and we continue to read the next chars.
Note : it is essential for this part that we continue the for loop because we dont want to execute the commands we have to be able to type the rest of the commands.

Now lets proceed to the consolewrite and how it is handling the match rewritings:



```c
int
consolewrite(struct inode *ip, char *buf, int n)
{
  int i;

  iunlock(ip);
  acquire(&cons.lock);


  if (buf[0] == '\t' && autocomplete_w) {
    autocomplete_w=0;
    input.tabr=input.r;
  }
  else if (buf[0]=='@'&&doubletab_detected){
    doubletab_detected=0;
    input.tabr=input.r;
  }
  else if (buf[0]!='@'&&buf[0]!='\t'&&doubletab_detected){
    char c = buf[0];
    consputc(c);
    input.buf[input.e++ % INPUT_BUF] = c;
  }
  else if (buf[0]=='@'&&!doubletab_detected){
    while (input.e > input.r) {
        delete_from_sequence(input.e-left_key_pressed_count);
        for(int i=0;i<input_sequence.size;i++)
          {
            if(input_sequence.data[i]>(input.e-left_key_pressed_count) % INPUT_BUF)
              input_sequence.data[i]--;
          }
      shift_buffer_left(0);
      consputc(BACKSPACE);
      input.e--;
    }
    cgaputc(' '); // an extra space for logic to match screen
    doubletab_detected=1;
    cgaputc('$'); cgaputc(' ');
  }
```

We have added a bunch of conditions to detect the auto complete matches or the incomplete command .

We have added two flags autocomplete_w and doubletab_detected they are flags to handle each case (signle tab or double tab)

`if (buf[0] == '\t' && autocomplete_w)`

- **Trigger:** The shell sends a `\t` character *while* the kernel is already in the middle of a single-match completion (`autocomplete_w` is true).
- **Interpretation:** This is the *second* `\t` of the `\t<completion>\t` message sent by `sh.c`. This signals the end of the completion word.
- **Action:**
- `autocomplete_w = 0;`: The single-match completion sequence is now finished. Reset the flag.
- `input.tabr = input.r;`: Resets the `tabr` pointer. This finalizes the state.

`else if (buf[0]=='@' && doubletab_detected)`

- **Trigger:** The shell sends an `@` character *while* the kernel is already in a double-tab state (`doubletab_detected` is true).
- **Interpretation:** This is the *second* `@` of the `@<prefix>@` message sent by `sh.c` after printing the match list. This signals the end of the prefix that needs to be re-displayed.
- **Action:**
- `doubletab_detected = 0;`: The double-tab sequence is over. Reset the flag.
- `input.tabr = input.r;`: Resets the `tabr` pointer.

`else if (buf[0]!='@' && buf[0]!='\t' && doubletab_detected)`

- **Trigger:** The shell sends a normal character (not `@` or `\t`) *while* the kernel is in a double-tab state.
- **Interpretation:** This is the user's original prefix (e.g., the 'l' from `ls`). The shell is sending it back to the kernel so it can be re-inserted into the input buffer and re-displayed on screen after the list of matches.
- **Action:**
- `consputc(c);`: Prints the character to the screen.
- `input.buf[input.e++ % INPUT_BUF] = c;`: "Stuffs" the character back into the kernel's input buffer, advancing the edit pointer `e`.

`else if (buf[0]=='@' && !doubletab_detected)`

- **Trigger:** The shell sends an `@` character, and we are *not* currently in a double-tab state.
- **Interpretation:** This is the *first* `@` of the `@<prefix>@` message. It's the shell's signal that a double-tab occurred, and the kernel should now prepare to receive and re-display the prefix.
- **Action:**
- `while (input.e > input.r)`: This loop **erases the user's current line from the screen and the buffer**. It repeatedly backspaces (`consputc(BACKSPACE)`) and decrements `input.e`. The complex logic with `delete_from_sequence` and `shift_buffer_left` suggests it's carefully managing an internal representation of the line to support advanced editing.
- `doubletab_detected = 1;`: Sets the flag to indicate we are now in a double-tab sequence.
- `cgaputc('$'); cgaputc(' ');`: Manually prints a new prompt (`$ `) to the screen, since the original line (including the prompt) was just erased.

`else if (buf[0]!='\t' && autocomplete_w)`

- **Trigger:** The shell sends a normal character (not `\t`) *while* the kernel is in a single-match completion state.
- **Interpretation:** This must be a character from the completed word (e.g., the 'l', then 's', then ' ' from `ls `) that `sh.c` is sending one by one.
- **Action:**
- This block acts like a normal character input handler.
- `consputc(c);`: Prints the character to the screen.
- `input.buf[input.e++ % INPUT_BUF] = c;`: "Stuffs" the completed character into the kernel's input buffer.
- The logic with `input_sequence` suggests it's also updating the state for your advanced line editor.

`else if (buf[0]=='\t' && !autocomplete_w)`

- **Trigger:** The shell sends a `\t` character, and we are *not* in a single-match completion state.
- **Interpretation:** This is the *first* `\t` of the `\t<completion>\t` message. It's the shell's signal that a single match was found, and the kernel should prepare to receive it.
- **Action:**

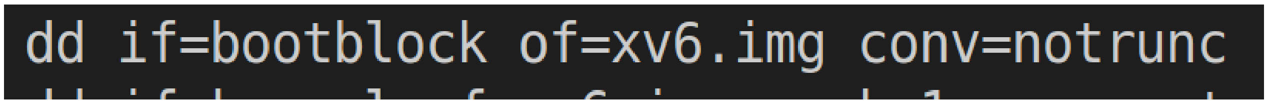
- `while (input.e > input.r)`: Just like in the double-tab case, this loop **erases the user's current partial word** from the screen and the buffer (e.g., erases the `l` that the user typed).
- `autocomplete_w = 1;`: Sets the flag to indicate we are now waiting for the characters of the completed word.

`else if(buf[0]!='\t' && !autocomplete_w)`

- **Trigger:** A character is written that is not a `\t`, and no completion sequence is active.
- **Interpretation:** This is the **default, normal behavior**. A program (like `echo` or `ls`) is just printing normal output to the console.
- **Action:**
- `for (i = 0; i < n; i++) consputc(buf[i] & 0xff);`: It simply loops through the provided buffer and prints every character to the screen. This is what the original `consolewrite` does.

Bootloader

12) The first (bootable) sector of the xv6 disk contains the contents of the bootblock file
The bootblock is created from the bootasm.S and bootmain.c files and contains the bootloader code that loads the kernel into memory.



```
dd if=bootblock of=xv6.img conv=notrunc
```

13)In xv6, most compiled files use the ELF (Executable and Linkable Format), which is a standard format for Unix systems. ELF files include sections like .text, .data, and .bss, along with headers that describe how to load the program into memory.However, the bootloader (bootblock) is not an ELF file — it is a raw binary file that contains only executable machine code. This is because the CPU cannot read ELF headers during boot.the BIOS only loads the first 512 bytes (one sector) from the disk.also its job is simple, to load kernel into memory in make file bootloader is made like this :$(OBJCOPY) -S -O binary -j .text bootblock.o bootblock

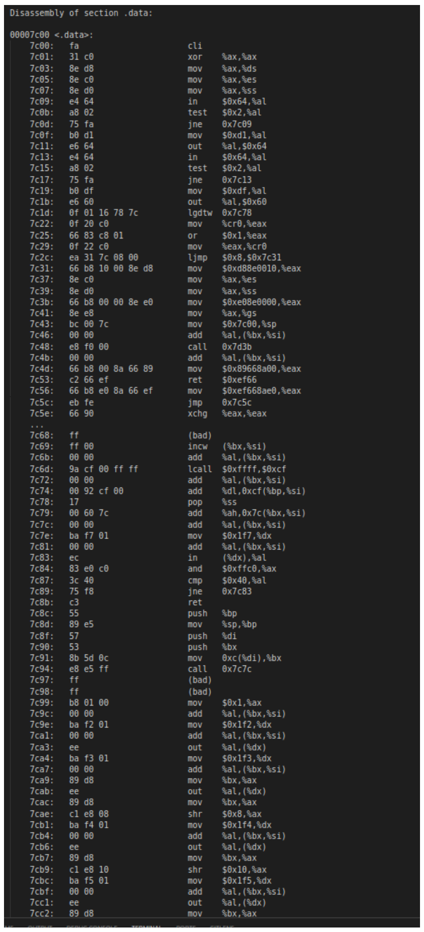
This copies only the .text section into a flat binary file without ELF headers.

To convert the bootloader into readable assembly, we use the following command:

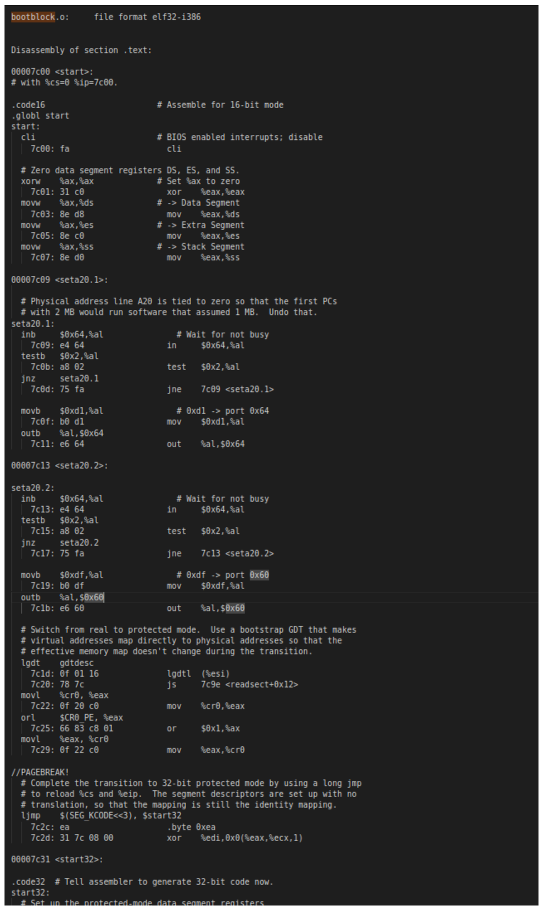
objdump -D -b binary -m i386 -M addr16,data16 --adjust-vma=0x7C00 bootblock > bootblock.S:

- -D : Disassemble all sections containing machine code (not just .text).
- -b binary: Input format is raw binary (not ELF or PE).
- -m i386: Architecture is Intel 80386 (we know xv6 runs on a 32 bit x86)
- -M addr16,data16: Use 16-bit addressing and data.Early boot code runs before the CPU switches to protected mode.
- --adjust-vma=0x7C00: Adjusts addresses to match the real bootloader's load address (0x7C00).
- Bootblock: Input file (raw machine code).
- > bootblock.S: Output disassembled result to a file named bootblock.S.

The output shows assembly instructions similar to those in **bootasm.S**, confirming that the bootblock contains the same bootloader code.



```
Disassembly of section .data:

00007c00 <.data>:
    7c00:   fa                      cli
    7c01:   31 c0                   xor    %ax,%ax
    7c03:   8e d8                   mov    %ax,%ds
    7c05:   8e c0                   mov    %ax,%es
    7c07:   8e d0                   mov    %ax,%ss
    7c09:   e4 64                   in     $0x64,%al
    7c0b:   a8 02                   test   $0x2,%al
    7c0d:   75 fa                   jne    0x7c09
    7c0f:   b0 d1                   mov    $0xd1,%al
    7c11:   e6 64                   out    %al,$0x64
    7c13:   e4 64                   in     $0x64,%al
    7c15:   a8 02                   test   $0x2,%al
    7c17:   75 fa                   jne    0x7c13
    7c19:   b0 df                   mov    $0xdf,%al
    7c1b:   e6 60                   out    %al,$0x60
    7c1d:   0f 01 16 78 7c          lgdtw  0x7c78
    7c22:   0f 20 c0                mov    %cr0,%eax
    7c25:   66 83 c8 01             or     $0x1,%eax
    7c29:   0f 22 c0                mov    %eax,%cr0
    7c2c:   ea 31 7c 08 00          ljmp   $0x8,$0x7c31
    7c31:   66 b8 10 00 8e d8       mov    $0xd88e0010,%eax
    7c37:   8e c0                   mov    %ax,%es
    7c39:   8e d0                   mov    %ax,%ss
    7c3b:   66 b8 00 00 8e e0       mov    $0xe08e0000,%eax
    7c41:   8e e8                   mov    %ax,%gs
    7c43:   bc 00 7c                mov    $0x7c00,%sp
    7c46:   00 00                   add    %al,(%bx,%si)
    7c48:   e8 f0 00                call   0x7d3b
    7c4b:   00 00                   add    %al,(%bx,%si)
    7c4d:   66 b8 00 8a 66 89       mov    $0x89668a00,%eax
    7c53:   c2 66 ef                ret    $0xef66
    7c56:   66 b8 e0 8a 66 ef       mov    $0xef668ae0,%eax
    7c5c:   eb fe                   jmp    0x7c5c
    7c5e:   66 90                   xchg   %eax,%eax
    ...
    7c68:   ff                      (bad)
    7c69:   ff 00                   incw   (%bx,%si)
    7c6b:   00 00                   add    %al,(%bx,%si)
    7c6d:   9a cf 00 ff ff          lcall  $0xffff,$0xcf
    7c72:   00 00                   add    %al,(%bx,%si)
    7c74:   00 92 cf 00             add    %dl,0xcf(%bp,%si)
    7c78:   17                      pop    %ss
    7c79:   00 60 7c                add    %ah,0x7c(%bx,%si)
    7c7c:   00 00                   add    %al,(%bx,%si)
    7c7e:   ba f7 01                mov    $0x1f7,%dx
    7c81:   00 00                   add    %al,(%bx,%si)
    7c83:   ec                      in     (%dx),%al
    7c84:   83 e0 c0                and    $0xffc0,%ax
    7c87:   3c 40                   cmp    $0x40,%al
    7c89:   75 f8                   jne    0x7c83
    7c8b:   c3                      ret
    7c8c:   55                      push   %bp
    7c8d:   89 e5                   mov    %sp,%bp
    7c8f:   57                      push   %di
    7c90:   53                      push   %bx
    7c91:   8b 5d 0c                mov    0xc(%di),%bx
    7c94:   e8 e5 ff                call   0x7c7c
    7c97:   ff                      (bad)
    7c98:   ff                      (bad)
    7c99:   b8 01 00                mov    $0x1,%ax
    7c9c:   00 00                   add    %al,(%bx,%si)
    7c9e:   ba f2 01                mov    $0x1f2,%dx
    7ca1:   00 00                   add    %al,(%bx,%si)
    7ca3:   ee                      out    %al,(%dx)
    7ca4:   ba f3 01                mov    $0x1f3,%dx
    7ca7:   00 00                   add    %al,(%bx,%si)
    7ca9:   89 d8                   mov    %bx,%ax
    7cab:   ee                      out    %al,(%dx)
    7cac:   89 d8                   mov    %bx,%ax
    7cae:   c1 e8 08                shr    $0x8,%ax
    7cb1:   ba f4 01                mov    $0x1f4,%dx
    7cb4:   00 00                   add    %al,(%bx,%si)
    7cb6:   ee                      out    %al,(%dx)
    7cb7:   89 d8                   mov    %bx,%ax
    7cb9:   c1 e8 10                shr    $0x10,%ax
    7cbc:   ba f5 01                mov    $0x1f5,%dx
    7cbf:   00 00                   add    %al,(%bx,%si)
    7cc1:   ee                      out    %al,(%dx)
    7cc2:   89 d8                   mov    %bx,%ax
```



```
bootblock.o:     file format elf32-i386


Disassembly of section .text:

00007c00 <start>:
# with %cs=0 %ip=7c00.

.code16                       # Assemble for 16-bit mode
.globl start
start:
  cli                         # BIOS enabled interrupts; disable
    7c00: fa                    cli

  # Zero data segment registers DS, ES, and SS.
  xorw    %ax,%ax             # Set %ax to zero
    7c01: 31 c0                 xor    %eax,%eax
  movw    %ax,%ds             # -> Data Segment
    7c03: 8e d8                 mov    %eax,%ds
  movw    %ax,%es             # -> Extra Segment
    7c05: 8e c0                 mov    %eax,%es
  movw    %ax,%ss             # -> Stack Segment
    7c07: 8e d0                 mov    %eax,%ss

00007c09 <seta20.1>:

  # Physical address line A20 is tied to zero so that the first PCs
  # with 2 MB would run software that assumed 1 MB.  Undo that.
seta20.1:
  inb     $0x64,%al               # Wait for not busy
    7c09: e4 64                 in     $0x64,%al
  testb   $0x2,%al
    7c0b: a8 02                 test   $0x2,%al
  jnz     seta20.1
    7c0d: 75 fa                 jne    7c09 <seta20.1>

  movb    $0xd1,%al               # 0xd1 -> port 0x64
    7c0f: b0 d1                 mov    $0xd1,%al
  outb    %al,$0x64
    7c11: e6 64                 out    %al,$0x64

00007c13 <seta20.2>:

seta20.2:
  inb     $0x64,%al               # Wait for not busy
    7c13: e4 64                 in     $0x64,%al
  testb   $0x2,%al
    7c15: a8 02                 test   $0x2,%al
  jnz     seta20.2
    7c17: 75 fa                 jne    7c13 <seta20.2>

  movb    $0xdf,%al               # 0xdf -> port 0x60
    7c19: b0 df                 mov    $0xdf,%al
  outb    %al,$0x60
    7c1b: e6 60                 out    %al,$0x60

  # Switch from real to protected mode.  Use a bootstrap GDT that makes
  # virtual addresses map directly to physical addresses so that the
  # effective memory map doesn't change during the transition.
  lgdt    gdtdesc
    7c1d: 0f 01 16              lgdtl  (%esi)
    7c20: 78 7c                 js     7c9e <readsect+0x12>
  movl    %cr0, %eax
    7c22: 0f 20 c0              mov    %cr0,%eax
  orl     $CR0_PE, %eax
    7c25: 66 83 c8 01           or     $0x1,%ax
  movl    %eax, %cr0
    7c29: 0f 22 c0              mov    %eax,%cr0

//PAGEBREAK!
  # Complete the transition to 32-bit protected mode by using a long jmp
  # to reload %cs and %eip.  The segment descriptors are set up with no
  # translation, so that the mapping is still the identity mapping.
  ljmp    $(SEG_KCODE<<3), $start32
    7c2c: ea                    .byte 0xea
    7c2d: 31 7c 08 00           xor    %edi,0x0(%eax,%ecx,1)

00007c31 <start32>:

.code32  # Tell assembler to generate 32-bit code now.
start32:
  # Set up the protected-mode data segment registers
```

14) objcopy is used to turn a ELF file into a raw binary so BIOS can directly load it into memory.

15)first stages of booting includes direct interactions with hardware and making the environment ready for the operating system. Assembly provides low-level access to hardware and can do tasks that are hardware specific, tasks such as setting up the stack, initializing registers, and switching the processor from real mode to protected mode.(a high level language such has c might have trouble doing them).

16)There are 4 main types for registers in x86 architecture:

1. General-Purpose Registers (GPR): These are used for various purposes by the command set. Example:

- Stack Pointer register (SP): Pointer to the top of the stack.
- Destination Index register (DI): Used as a pointer to a destination in stream operations.

2. Segment Registers:

- Code Segment (CS): Points to the segment containing the current program code
- Stack Segment (SS): Points to the segment containing the stack.

3.The flag register:

- Carry Flag (CF): Set if an arithmetic operation results in a carry (for addition) or borrow (for subtraction).
- Zero Flag (ZF): Set if an operation results in zero.

4. Control Register:

- CR2: The CR2 register is used in page-fault exception processing.(It contains the linear address that caused a page fault.)
- CR3: when virtual memory is enabled, and it holds the physical address of the Page Directory.

17) a x86 processor starts in a mode called "real mode" during booting . This is a 16-bit mode which has backward compatibility with older x86 processors. but it has limitations, for example it has a maximum of 1MB of addressable memory. Modern operating systems quickly switch the processor to “protected mode”, which supports 32-bit addressing and has access to features like virtual memory. In 64-bit systems, the processor is switched to “long mode” for 64-bit support.

18)purpose: Protected mode provides memory protection and allows the CPU to manage virtual memory. It ensures that programs cannot access memory regions they are not allowed to. It also has features like multitasking and paging.
CPUs start in real mode during boot, which has full access to hardware(The bootloader runs here).After the bootloader sets up necessary structures, the kernel switches the CPU to protected mode. In this mode, logical (virtual) addresses used by programs are mapped to physical memory through tables managed by the CPU.

19)

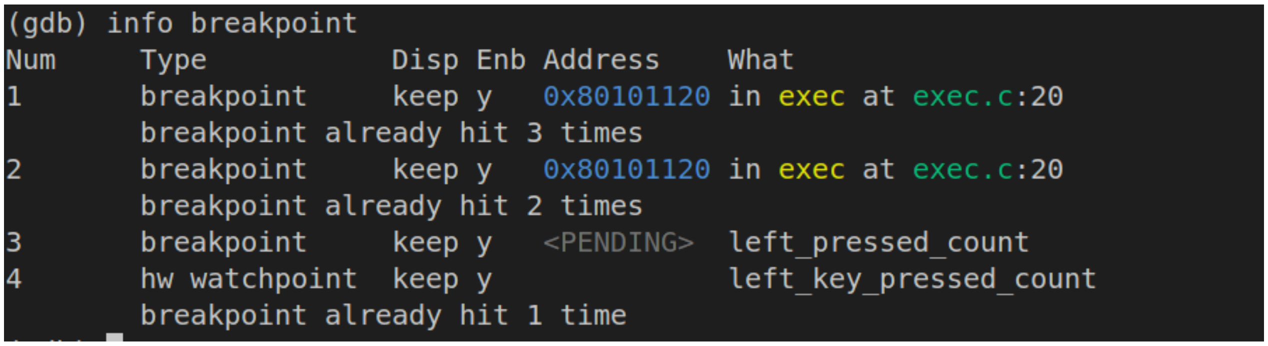
1.The first 1 MB of memory (0x00000–0xFFFFF) is used by the BIOS, interrupt vectors, and real-mode data structures. Placing the kernel there could overwrite essential system data.

2. The kernel runs in protected mode, which expects the code and data to be in a high memory region separate from real-mode structures.

3. 0x100000 is a common location in x86 systems for loading kernels because working with it is pretty straightforward and there is no need to do hard calculations.(for memory management)

Debugging

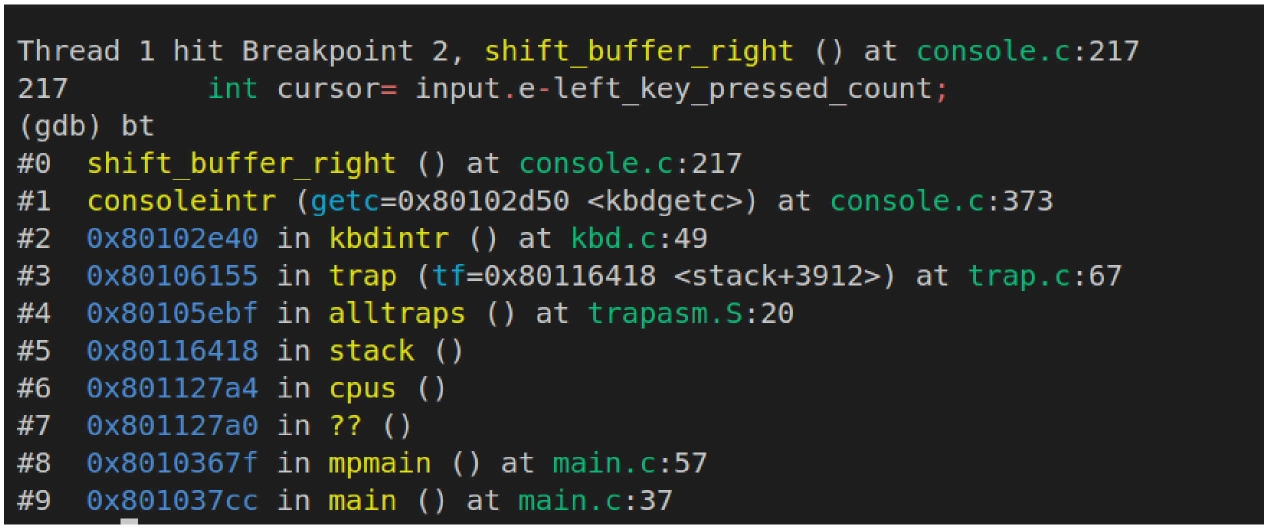
20) for viewing all the breakpoints we can simply use “info breakpoint”, we can also use “ break x” where x can be a line number, function name, *location , file name:line number , it can also be conditional.



```
(gdb) info breakpoint
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x80101120 in exec at exec.c:20
        breakpoint already hit 3 times
2       breakpoint     keep y   0x80101120 in exec at exec.c:20
        breakpoint already hit 2 times
3       breakpoint     keep y   <PENDING>  left_pressed_count
4       hw watchpoint  keep y              left_key_pressed_count
        breakpoint already hit 1 time
```

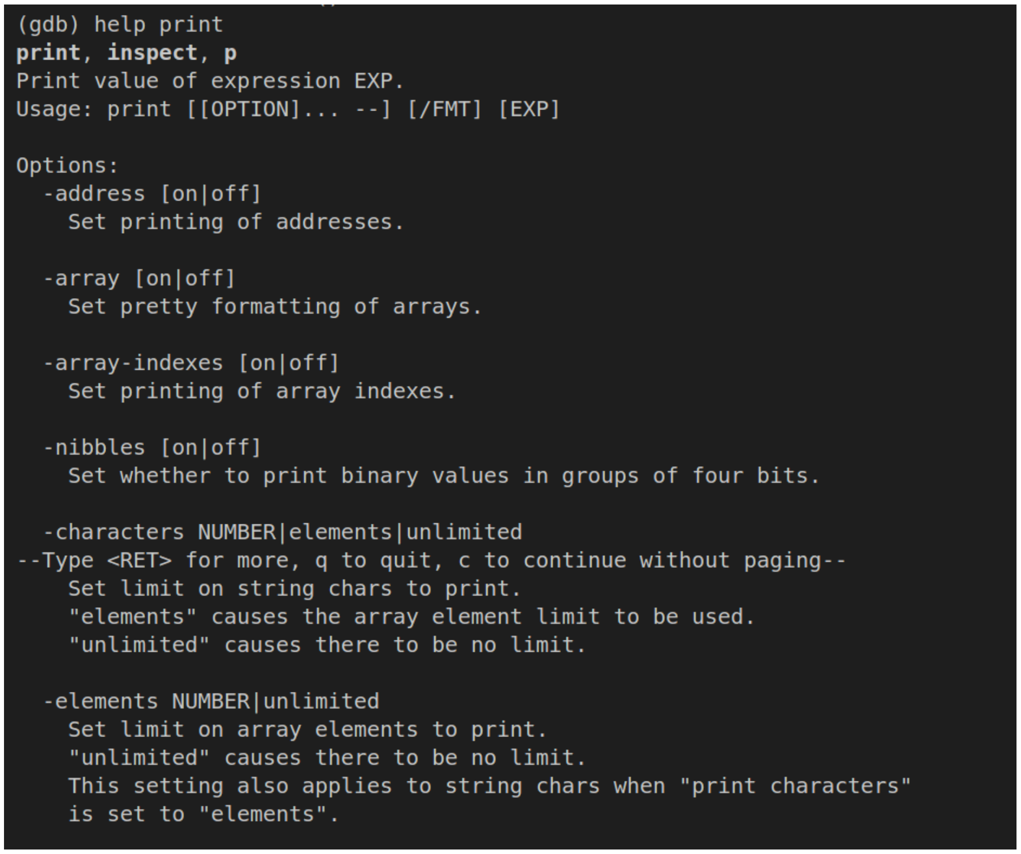
21) for deleting a break point we can use “delete x” where x is the number of breakpoint (which can be seen in the picture above)

22) bt command shows the call stack, it basically shows what functions were called to get to this point. So for example here it first calls main then goes into mpmain and so on… .

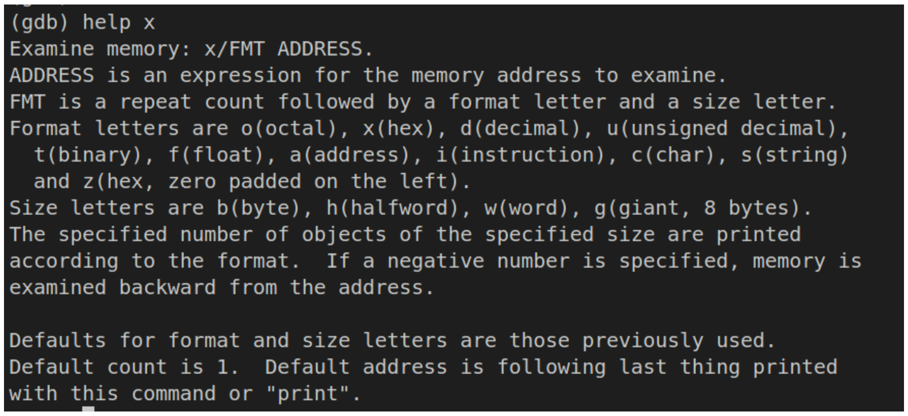


```
Thread 1 hit Breakpoint 2, shift_buffer_right () at console.c:217
217         int cursor= input.e-left_key_pressed_count;
(gdb) bt
#0  shift_buffer_right () at console.c:217
#1  consoleintr (getc=0x80102d50 <kbdgetc>) at console.c:373
#2  0x80102e40 in kbdintr () at kbd.c:49
#3  0x80106155 in trap (tf=0x80116418 <stack+3912>) at trap.c:67
#4  0x80105ebf in alltraps () at trapasm.S:20
#5  0x80116418 in stack ()
#6  0x801127a4 in cpus ()
#7  0x801127a0 in ?? ()
#8  0x8010367f in mpmain () at main.c:57
#9  0x801037cc in main () at main.c:37
```

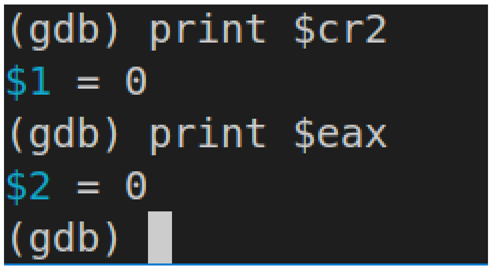
23)



```
(gdb) help print
print, inspect, p
Print value of expression EXP.
Usage: print [[OPTION]... --] [/FMT] [EXP]

Options:
  -address [on|off]
    Set printing of addresses.

  -array [on|off]
    Set pretty formatting of arrays.

  -array-indexes [on|off]
    Set printing of array indexes.

  -nibbles [on|off]
    Set whether to print binary values in groups of four bits.

  -characters NUMBER|elements|unlimited
--Type <RET> for more, q to quit, c to continue without paging--
    Set limit on string chars to print.
    "elements" causes the array element limit to be used.
    "unlimited" causes there to be no limit.

  -elements NUMBER|unlimited
    Set limit on array elements to print.
    "unlimited" causes there to be no limit.
    This setting also applies to string chars when "print characters"
    is set to "elements".
```

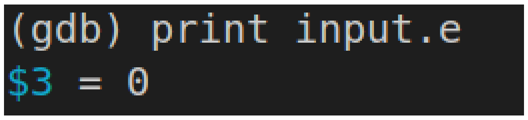


```
(gdb) help x
Examine memory: x/FMT ADDRESS.
ADDRESS is an expression for the memory address to examine.
FMT is a repeat count followed by a format letter and a size letter.
Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal),
  t(binary), f(float), a(address), i(instruction), c(char), s(string)
  and z(hex, zero padded on the left).
Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes).
The specified number of objects of the specified size are printed
according to the format.  If a negative number is specified, memory is
examined backward from the address.

Defaults for format and size letters are those previously used.
Default count is 1.  Default address is following last thing printed
with this command or "print".
```



```
(gdb) print $cr2
$1 = 0
(gdb) print $eax
$2 = 0
(gdb)
```

Print: prints the value of a variable or expression



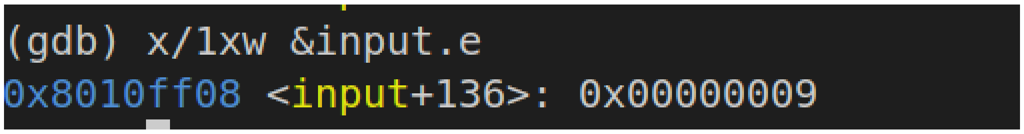
```
(gdb) print input.e
$3 = 0
```

X:

This is to examine memory and it is used like this:

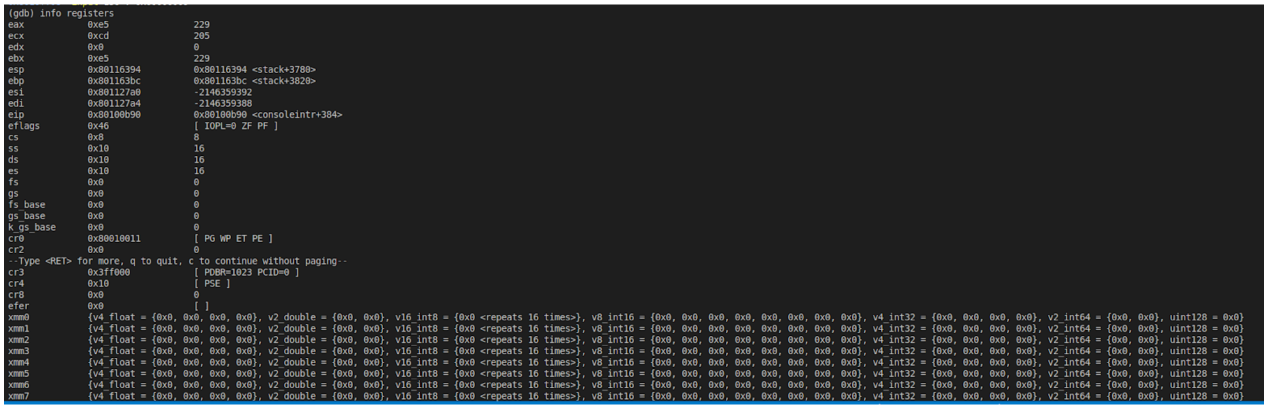
x/(count)(format)(size) address

So for example here we got count 1 and it to show it in format hex also the size is word(here we show one word which is the location of input e, also the argument of x is address.



```
(gdb) x/1xw &input.e
0x8010ff08 <input+136>: 0x00000009
```

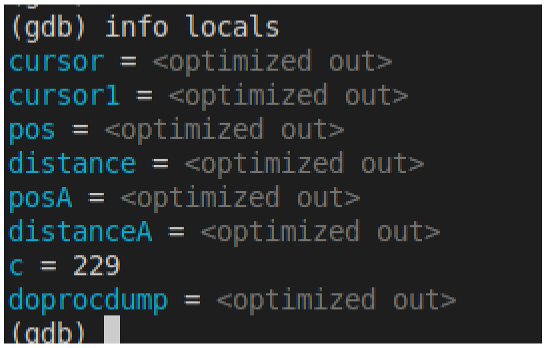
24) we can show registers info with the command “info registers”



```
(gdb) info registers
eax            0xe5                229
ecx            0xcd                205
edx            0x0                 0
ebx            0xe5                229
esp            0x80116394          0x80116394 <stack+3780>
ebp            0x801163bc          0x801163bc <stack+3820>
esi            0x801127a0          -2146359392
edi            0x801127a4          -2146359388
eip            0x80100b90          0x80100b90 <consoleintr+384>
eflags         0x46                [ IOPL=0 ZF PF ]
cs             0x8                 8
ss             0x10                16
ds             0x10                16
es             0x10                16
fs             0x0                 0
gs             0x0                 0
fs_base        0x0                 0
gs_base        0x0                 0
k_gs_base      0x0                 0
cr0            0x80010011          [ PG WP ET PE ]
cr2            0x0                 0
--Type <RET> for more, q to quit, c to continue without paging--
cr3            0x3ff000            [ PDBR=1023 PCID=0 ]
cr4            0x10                [ PSE ]
cr8            0x0                 0
efer           0x0                 [ ]
xmm0           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm1           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm2           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm3           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm4           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm5           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm6           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm7           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
```

Also we can see the local values of current functions(where we hit breakpoint) using “info locals”.

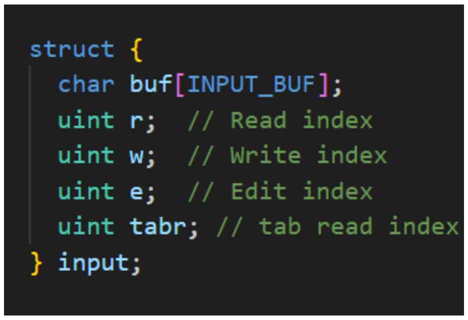
(we stopped where right arrow was implemented, optimized out means compiler removed the value of the variable in optimization)



```
(gdb) info locals
cursor = <optimized out>
cursor1 = <optimized out>
pos = <optimized out>
distance = <optimized out>
posA = <optimized out>
distanceA = <optimized out>
c = 229
doprocdump = <optimized out>
(gdb)
```

Esi and edi are mainly used for memory operations and generally where we need a source and a destination, esi points to the source and edi points to the destination.(GPR)

25)



```
struct {
  char buf[INPUT_BUF];
  uint r;  // Read index
  uint w;  // Write index
  uint e;  // Edit index
  uint tabr; // tab read index
} input;
```

The struct input is the heart of the console driver’s line discipline. Its primary purpose is to act as a **kernel-side buffer** for characters typed by the user on the keyboard. This decouples the speed of user typing from the speed at which a user-space program (like the shell) reads the input. This is a classic producer-consumer problem, where the keyboard interrupt handler (consoleintr) is the "producer" and the console read function (consoleread) is the "consumer".

`char buf[INPUT_BUF]`

This is the raw storage array for characters. It holds everything the user types before it's read by a program. Because it's a circular buffer, when an index goes past the end (`INPUT_BUF - 1`), it wraps back around to 0. This is handled using the modulo operator (`% INPUT_BUF`).

`uint r` (Read Index)

This index points to the next character in `buf` that will be given to a process when `consoleread` is called. It represents the start of the "unconsumed" data that has already been committed. It is the "consumer's" pointer.

`uint w` (Write/Commit Index)

Role: This index points to the position *after* the last character of a fully committed line. A line is "committed" when the user presses **Enter** (`\n`) or **Ctrl-D**. The key condition `r != w` signifies that there is at least one complete line in the buffer ready to be read by a program.

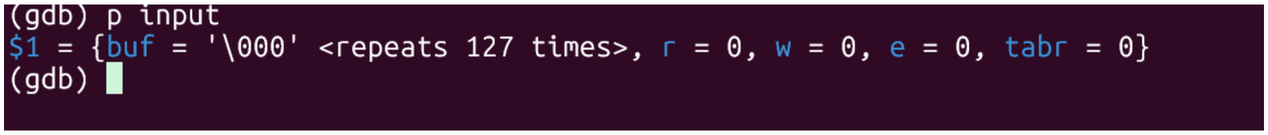
`uint e` (Edit Index)

This is the most active index. It points to the position *after* the last character currently being typed on the command line. It moves forward as the user types characters and backward as they press Backspace. The characters between `w` and `e` represent the current, uncommitted line that the user is editing.

`uint tabr` (Tab Read Index - Your Custom Field)
Role: Based on your `consolewrite` logic, this variable acts as a temporary "save point" for the `r` index. During your complex tab-completion protocol, the buffer is heavily manipulated. `tabr` likely helps the

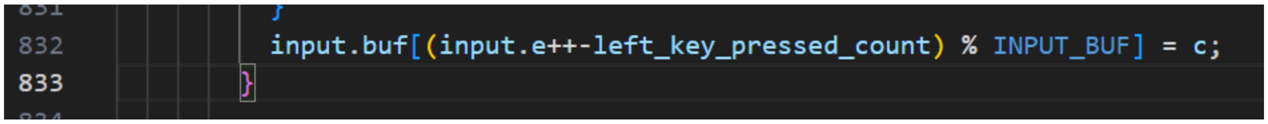
driver remember where the valid, readable data began before the tab sequence started, ensuring the buffer state can be correctly restored or finalized after completion.

Now in GDP we can print the the insput structure any time by writing print input
Here is a input description before adding anything to it:



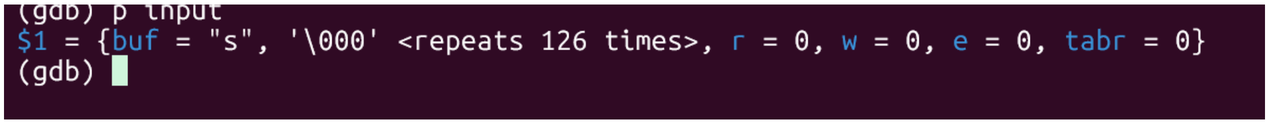
```
(gdb) p input
$1 = {buf = '\000' <repeats 127 times>, r = 0, w = 0, e = 0, tabr = 0}
(gdb) 
```

As you can see the buffer is empty and all of the pointers are set to 0.
Now we add a pointer to the line 833 of console.c which is where its incrementing the input.e when a normal char is pressed:



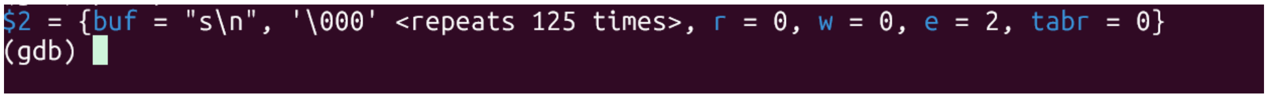
```
832                input.buf[(input.e++-left_key_pressed_count) % INPUT_BUF] = c;
833            }
```

Then we continue and press a char like ‘s’:



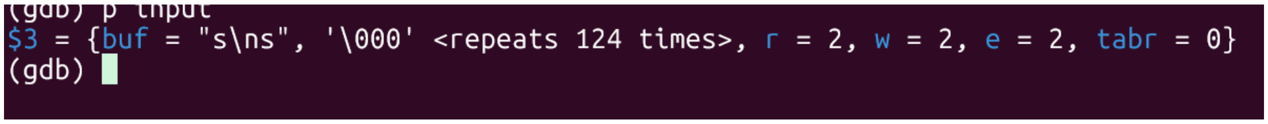
```
(gdb) p input
$1 = {buf = "s", '\000' <repeats 126 times>, r = 0, w = 0, e = 0, tabr = 0}
(gdb) 
```

As you can see the buffer now is showing s but the e is still 0 this is because the input++ the increment happens after pushing to the buffer so the breakpoint prevents the program to do that no if i add another break point at the end of the default case of the consintr it waits until i press \n this way we can observe how the input e is being upddated :



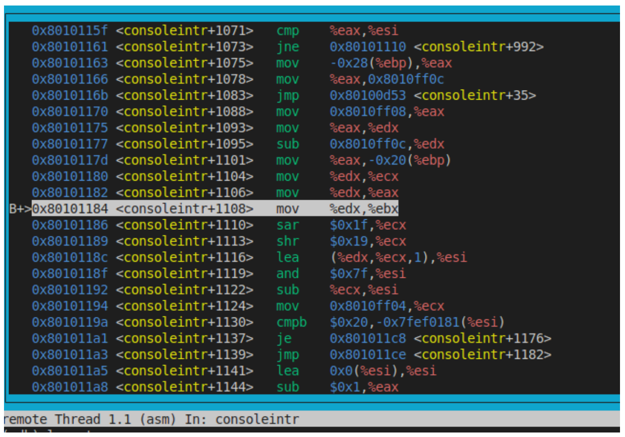
```
$2 = {buf = "s\n", '\000' <repeats 125 times>, r = 0, w = 0, e = 2, tabr = 0}
(gdb) 
```

Also if we move the breakpoint to the part where the \n is being handled we can see that the input r and input w are correctly getting updated to input e:

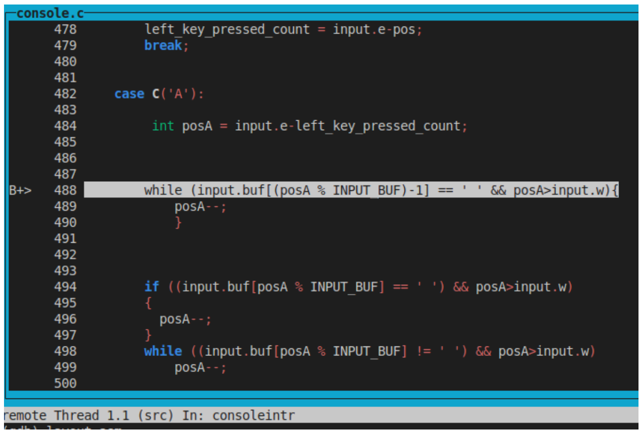


```
(gdb) p input
$3 = {buf = "s\ns", '\000' <repeats 124 times>, r = 2, w = 2, e = 2, tabr = 0}
(gdb) 
```

26)layout asm shows the part of the code that the breakpoint happened in assembly and layout src shows it in c.

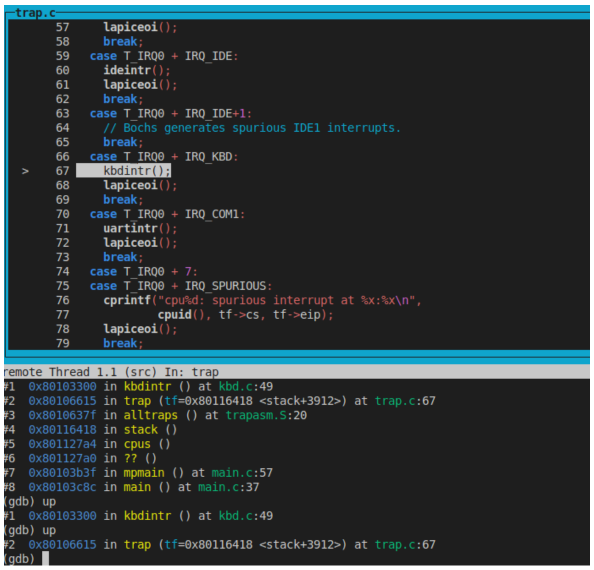


```
    0x8010115f <consoleintr+1071>   cmp    %eax,%esi
    0x80101161 <consoleintr+1073>   jne    0x80101110 <consoleintr+992>
    0x80101163 <consoleintr+1075>   mov    -0x28(%ebp),%eax
    0x80101166 <consoleintr+1078>   mov    %eax,0x8010ff0c
    0x8010116b <consoleintr+1083>   jmp    0x80100d53 <consoleintr+35>
    0x80101170 <consoleintr+1088>   mov    0x8010ff08,%eax
    0x80101175 <consoleintr+1093>   mov    %eax,%edx
    0x80101177 <consoleintr+1095>   sub    0x8010ff0c,%edx
    0x8010117d <consoleintr+1101>   mov    %eax,-0x20(%ebp)
    0x80101180 <consoleintr+1104>   mov    %edx,%ecx
    0x80101182 <consoleintr+1106>   mov    %edx,%eax
B+> 0x80101184 <consoleintr+1108>   mov    %edx,%ebx
    0x80101186 <consoleintr+1110>   sar    $0x1f,%ecx
    0x80101189 <consoleintr+1113>   shr    $0x19,%ecx
    0x8010118c <consoleintr+1116>   lea    (%edx,%ecx,1),%esi
    0x8010118f <consoleintr+1119>   and    $0x7f,%esi
    0x80101192 <consoleintr+1122>   sub    %ecx,%esi
    0x80101194 <consoleintr+1124>   mov    0x8010ff04,%ecx
    0x8010119a <consoleintr+1130>   cmpb   $0x20,-0x7fef0181(%esi)
    0x801011a1 <consoleintr+1137>   je     0x801011c8 <consoleintr+1176>
    0x801011a3 <consoleintr+1139>   jmp    0x801011ce <consoleintr+1182>
    0x801011a5 <consoleintr+1141>   lea    0x0(%esi),%esi
    0x801011a8 <consoleintr+1144>   sub    $0x1,%eax
remote Thread 1.1 (asm) In: consoleintr
```



```
console.c
            left_key_pressed_count = input.e-pos;
            break;


        case C('A'):

             int posA = input.e-left_key_pressed_count;



B+>         while (input.buf[(posA % INPUT_BUF)-1] == ' ' && posA>input.w){
                posA--;
                }



            if ((input.buf[posA % INPUT_BUF] == ' ') && posA>input.w)
            {
              posA--;
            }
            while ((input.buf[posA % INPUT_BUF] != ' ') && posA>input.w)
                posA--;

remote Thread 1.1 (src) In: consoleintr
```

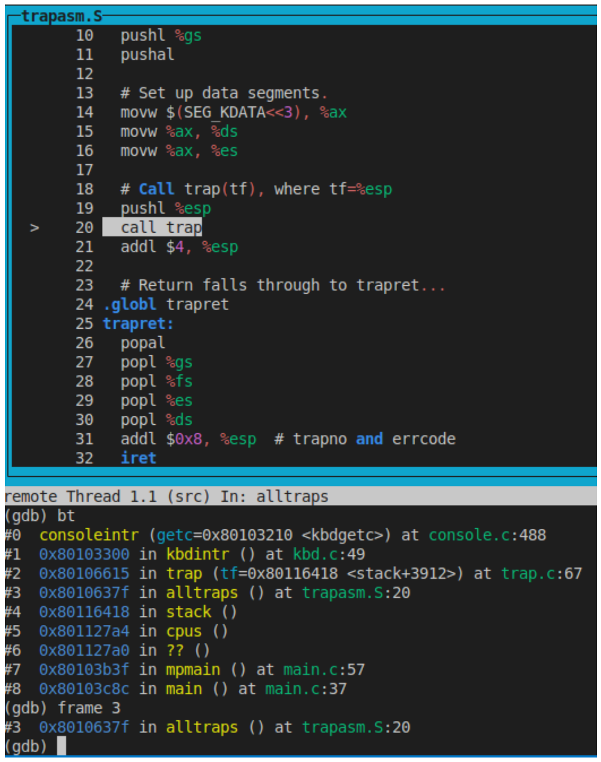
27) we can use up and down command to move up and down in the call stack



```
trap.c
      lapiceoi();
      break;
    case T_IRQ0 + IRQ_IDE:
      ideintr();
      lapiceoi();
      break;
    case T_IRQ0 + IRQ_IDE+1:
      // Bochs generates spurious IDE1 interrupts.
      break;
    case T_IRQ0 + IRQ_KBD:
>     kbdintr();
      lapiceoi();
      break;
    case T_IRQ0 + IRQ_COM1:
      uartintr();
      lapiceoi();
      break;
    case T_IRQ0 + 7:
    case T_IRQ0 + IRQ_SPURIOUS:
      cprintf("cpu%d: spurious interrupt at %x:%x\n",
              cpuid(), tf->cs, tf->eip);
      lapiceoi();
      break;
remote Thread 1.1 (src) In: trap
#1  0x80103300 in kbdintr () at kbd.c:49
#2  0x80106615 in trap (tf=0x80116418 <stack+3912>) at trap.c:67
#3  0x8010637f in alltraps () at trapasm.S:20
#4  0x80116418 in stack ()
#5  0x801127a4 in cpus ()
#6  0x801127a0 in ?? ()
#7  0x80103b3f in mpmain () at main.c:57
#8  0x80103c8c in main () at main.c:37
(gdb) up
#1  0x80103300 in kbdintr () at kbd.c:49
(gdb) up
#2  0x80106615 in trap (tf=0x80116418 <stack+3912>) at trap.c:67
(gdb) 
```

For example here we moved up 2 times to reach the part where kbdintr was called in trap.

We can also use frame to jump to an specific index in the stack.



```
trapasm.S
      pushl %gs
      pushal

      # Set up data segments.
      movw $(SEG_KDATA<<3), %ax
      movw %ax, %ds
      movw %ax, %es

      # Call trap(tf), where tf=%esp
      pushl %esp
>     call trap
      addl $4, %esp

      # Return falls through to trapret...
    .globl trapret
    trapret:
      popal
      popl %gs
      popl %fs
      popl %es
      popl %ds
      addl $0x8, %esp  # trapno and errcode
      iret

remote Thread 1.1 (src) In: alltraps
(gdb) bt
#0  consoleintr (getc=0x80103210 <kbdgetc>) at console.c:488
#1  0x80103300 in kbdintr () at kbd.c:49
#2  0x80106615 in trap (tf=0x80116418 <stack+3912>) at trap.c:67
#3  0x8010637f in alltraps () at trapasm.S:20
#4  0x80116418 in stack ()
#5  0x801127a4 in cpus ()
#6  0x801127a0 in ?? ()
#7  0x80103b3f in mpmain () at main.c:57
#8  0x80103c8c in main () at main.c:37
(gdb) frame 3
#3  0x8010637f in alltraps () at trapasm.S:20
(gdb)
```