CS 140 Lab Report 08

Dale Sealtiel T. Flores
2023-11373
THX/WXY

1. Show all relevant changes made (with corresponding filenames) via code screenshots or snippets for the exercise in Section 2.8, briefly describe what each change does. Ensure that your changes are properly committed and pushed to your Github Classroom repository.

Answer: These are the changes that I made in order to implement the needed behavior in Section 2.8

```
static const unsigned char keys[] = {
  0x1b, 0x5b, 0x41, // up
  0x1b, 0x5b, 0x41, // up
  0x1b, 0x5b, 0x42, // down
  0x1b, 0x5b, 0x42, // down
  0x1b, 0x5b, 0x44, // left
  0x1b, 0x5b, 0x43, // right
  0x1b, 0x5b, 0x44, // left
  0x1b, 0x5b, 0x43, // right
  'b','a', C('X')   // b a ctrl-x
};
```

Figure 1: Change 1 in `console.c`

This change is to have the key sequence be put in a sequence in order to have its order preserved **(Up, Up, Down, Down, Left, Right, Left, Right, b, a, and then Ctrl-x).**

```
static int index = 0;
```

Figure 2: Change 2 in `console.c`

This change is to keep track of the key presses to be used in *figure 1.*

```
unsigned char uc = (unsigned char)c;
```

Figure 3: Change 3 in `console.c`

This change is needed since if we look at `consoleintr` it passes in an `int c`. We need to convert it first in order to know if we pressed the right button.

```
if (uc == keys[index]) {
  index++;
  if (index == 27) {
    (*(volatile uint32 *)0x100000) = 0x5555;
  }
} else{
    if (uc == keys[0]) {
      index = 1;
    } else {
      index = 0;
    }
}
```

Figure 4: Change 4 in `console.c`

This is the main change for the behavior needed for Section 2.8 It checks whether the key pressed is in the key sequence in *figure 1*. If it is, it will increment the global index. If not, it will reset the global index back to 0. If the global index reaches the end of the sequence, it will shutdown xv6. The code that was used was copy and pasted from `sys_shutdown` in `kernel/syscall.c`.

2. Describe what happens when `WriteReg(IER, IER_TX_ENABLE | IER_RX_ENABLE);` in `uartinit` is commented out and explain why that is the case.

Answer: The user cannot type in the console of xv6. This is explained in Section 2.6 of the Lab Exercise specifications(*"When the UART becomes ready to either read or write data, it raises an interrupt"*). We can also get hints from the comments in `uartinit` above the instruction said in the question (*"enable transmit and receive interrupts"*). Now we can conclude that since we disabled interrupts by commenting out the instruction, the UART is now unable to read or write the data needed for typing in the console.

3. Identify the block of xv6 code (and which file it is in) that allows the kernel to map the page frame containing physical address `0x10000000` to a virtual page

Answer: The code that allows the kernel to map the page frame containing the physical address to a virtual page can be found in `pagetable_t kvmmake(void)` inside of `kernel/vm.c`

```
// Make a direct-map page table for the kernel.
pagetable_t
kvmmake(void)
{
  pagetable_t kpgtbl;

  kpgtbl = (pagetable_t) kalloc();
  memset(kpgtbl, 0, PGSIZE);

  // Shutdown device
  kvmmap(kpgtbl, 0x100000, 0x100000, PGSIZE, PTE_R | PTE_W);
```

Figure 5: Code for mapping the physical address to a virtual page

4. For the following items, refer to `uartputc_sync` in `kernel/uart.c`:
   (a) Explain how `uartputc` and `uartputc_sync` differ in terms of how the circular transmit buffer and how interrupts are used.

Answer: The difference of `uartputc` and `uartputc_sync` in their usage of the circular transmit buffer and interrupts is `uartputc` uses them while `uartputc_sync` doesn't. We can see from the comments in `kernel/uart.c`,

```
// alternate version of uartputc() that doesn't
// use interrupts, for use by kernel printf() and
// to echo characters. it spins waiting for the uart's
// output register to be empty.
void
uartputc_sync(int c)
{
  if(panicking == 0)
    push_off();

  if(panicked){
    for(;;)
      ;
  }

  // wait for Transmit Holding Empty to be set in LSR.
  while((ReadReg(LSR) & LSR_TX_IDLE) == 0)
    ;
  WriteReg(THR, c);

  if(panicking == 0)
    pop_off();
}
```

Figure 6: Comments in `uartputc_sync` in `kernel/uart.c`

`uartputc_sync` doesn't use interrupts. We can also observe from the code itself that it doesn't use the circular transmit buffer. It does `WriteReg(THR, c)` instead of using the variables `uart_tx_r` and `uart_tx_w` to write to the buffer.

```
// add a character to the output buffer and tell the
// UART to start sending if it isn't already.
// blocks if the output buffer is full.
// because it may block, it can't be called
// from interrupts; it's only suitable for use
// by write().
void
uartputc(int c)
{
  if(panicking == 0)
    acquire(&uart_tx_lock);

  if(panicked){
    for(;;)
      ;
  }
  while(uart_tx_w == uart_tx_r + UART_TX_BUF_SIZE){
    // buffer is full.
    // wait for uartstart() to open up space in the buffer.
    sleep(&uart_tx_r, &uart_tx_lock);
  }
  uart_tx_buf[uart_tx_w % UART_TX_BUF_SIZE] = c;
  uart_tx_w += 1;
  uartstart();
  if(panicking == 0)
    release(&uart_tx_lock);
}
```

Figure 7: Code of `uartputc` in `kernel/uart.c`

(b) Excluding those in `kernel/printf.c`, enumerate all instances in the xv6 kernel that call `consputc` (which calls `uartputc_sync`) by showing screenshots or code snippets and their respecting files.

Answer: There are **3** instances of `consputc` (*excluding those in* `kernel/printf.c`). All of the calls are from `consoleintr` in `kernel/console.c`

```
case C('U'):  // Kill line.
  while(cons.e != cons.w &&
        cons.buf[(cons.e-1) % INPUT_BUF_SIZE] != '\n'){
    cons.e--;
    consputc(BACKSPACE);
  }
  break;
```

Figure 8: `constput(BACKSPACE)` in `kernel/console.c`

```
case C('H'): // Backspace
case '\x7f': // Delete key
  if(cons.e != cons.w){
    cons.e--;
    consputc(BACKSPACE);
  }
  break;
```

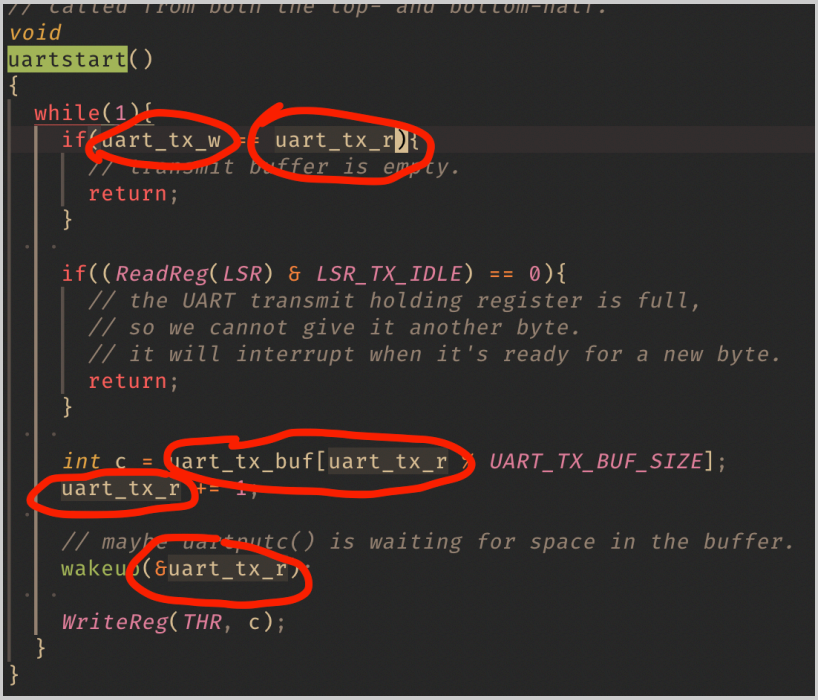
Figure 9: `consputc(BACKSPACE)` in `kernel/console.c`

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

    // echo back to the user.
    consputc(c);

    // store for consumption by consoleread().
    cons.buf[cons.e++ % INPUT_BUF_SIZE] = c;

    if(c == '\n' || c == C('D') || cons.e-cons.r == INPUT_BUF_SIZE){
      // wake up consoleread() if a whole line (or end-of-file)
      // has arrived.
      cons.w = cons.e;
      wakeup(&cons.r);
    }
  }
  break;
```

Figure 10: `consputc(c)` in `kernel/console.c`

5. Explain why there is a need for callers of `uartstart` to obtain the `uart_tx_lock` lock.

Answer: If we look at the code of `uartstart`, it uses the circular transmit buffer.



```
// called from both the top- and bottom-half.
void
uartstart()
{
  while(1){
    if(uart_tx_w == uart_tx_r){
      // transmit buffer is empty.
      return;
    }

    if((ReadReg(LSR) & LSR_TX_IDLE) == 0){
      // the UART transmit holding register is full,
      // so we cannot give it another byte.
      // it will interrupt when it's ready for a new byte.
      return;
    }

    int c = uart_tx_buf[uart_tx_r % UART_TX_BUF_SIZE];
    uart_tx_r += 1;

    // maybe uartputc() is waiting for space in the buffer.
    wakeup(&uart_tx_r);

    WriteReg(THR, c);
  }
}
```

Figure 11: `uartstart` code block

These variables are also used in different functions since it is the variables of the circular transmit buffer. Since these variables are used in different functions such as `uartputc`, having locks ensures that it doesn't cause race conditions ensuring that the data in the circular buffer is consistent.

6. Identify the functions called in sequence starting from `devintr` and until `consputc` that results in a key such as `a` that that the user presses to be displayed on screen. Do not include macro "calls".

Answer: When a user types a character such as `a` it will raise an interrupt that activates the trap handler which calls `devintr` which it will call `r_scause()` then it will call `int irq = plic_claim()` to determine which device interrupted. Since it is it the UART, it will now call `uartintr` which will read the input which will then call `consoleintr` which will then call `consput(c)` which will display the character that was typed which is `a` for our case.

Table 1: Summary

Order	Function
1	`devintr`
2	`r_scause()`
3	`int irq = plic_claim()`
4	`uartintr`
5	`consoleintr`
6	`consput(c)`

7. Supposing that `uart_tx_buf` is currently set to `140`, illustrate what happens to the buffer step-by-step when `uartputc('W');` is called which causes the buffer to become full. Assume that the call

to `uartstart` by `uartputc('W')` causes the buffer to be fully processed (i.e., it will be empty when `uartstart` returns). Ensure all data read from and written to the UART is shown.

Answer: We know that that buffer size is 32 bytes. So since `uart_tx_buf` is set to `140` and calling `uartputc('W')` will cause it to become full. We have $140 - 31 = 109$ for the `uart_tx_r`. In summary we have `uart_tx_w` $= 140$ and `uart_tx_r` $= 109$.
After `uartstart` is called, it will now process the buffer.

Step	`uart_tx_r`	`uart_tx_r` % 32	State
1	109	13	`uart_tx_buf[13]`
2	110	14	`uart_tx_buf[14]`
3	111	15	`uart_tx_buf[15]`
4	112	16	`uart_tx_buf[16]`
5	113	17	`uart_tx_buf[17]`
6	114	18	`uart_tx_buf[18]`
7	115	19	`uart_tx_buf[19]`
8	116	20	`uart_tx_buf[20]`
9	117	21	`uart_tx_buf[21]`
10	118	22	`uart_tx_buf[22]`
11	119	23	`uart_tx_buf[23]`
12	120	24	`uart_tx_buf[24]`
13	121	25	`uart_tx_buf[25]`
14	122	26	`uart_tx_buf[26]`
15	123	27	`uart_tx_buf[27]`
16	124	28	`uart_tx_buf[28]`
17	125	29	`uart_tx_buf[29]`
18	126	30	`uart_tx_buf[30]`
19	127	31	`uart_tx_buf[31]`
20	128	0	`uart_tx_buf[0]`
21	129	1	`uart_tx_buf[1]`
22	130	2	`uart_tx_buf[2]`
23	131	3	`uart_tx_buf[3]`
24	132	4	`uart_tx_buf[4]`
25	133	5	`uart_tx_buf[5]`
26	134	6	`uart_tx_buf[6]`
27	135	7	`uart_tx_buf[7]`
28	136	8	`uart_tx_buf[8]`
29	137	9	`uart_tx_buf[9]`
30	138	10	`uart_tx_buf[10]`
31	139	11	`uart_tx_buf[11]`
32	140	12	`uart_tx_buf[12]`

After `uartstart` finishes, both `uart_tx_r` and `uart_tx_w` will be `141`.

8. Illustrate how the xv6 UART driver ensures that writing to relatively slow uart devices (i.e, a large number of C instructions can be executed before the UART is ready for another write) does not hamper program performance. Assume that the circular buffer does not get full.

Answer: In xv6, it uses a circular buffer in order to not hamper program performance when writing to slow UART devices. When the UART is not ready, it will store the data to to the circular buffer. When it is ready, it will then call `uartstart` to then process the buffer. This way, the CPU can continue executing while waiting for it to be ready.