CS 344 : Operating Systems Lab **Lab Assignment 2**

Group Number: 28

Tanay Maheshwari	190101092
Shreyansh Meena	190101084
Vignesh Ravichandra Rao	190101109
Nikhil Kumar Pandey	190123040

Commands to reproduce changes:
The patch file `G28_lab2a.patch` is included in the submission. This patch file works on the entire (unedited) XV6 source directory. Assuming that the XV6 source directory on your machine is `xv6-public`, run the following command from the parent directory:

```
patch -uN -d xv6-public < G28_lab2a.patch
```

In case the name of the XV6 source code directory is different, replace `xv6-public` in the above command with the appropriate directory name. **Note**: This patch is for both Task 1 and Task 2.

Task 1.1) Caret Navigation

Observations regarding `console.c`.

- The function `cgaputc` is responsible for printing a character onto the CGA (Color Graphics Adapter) of the emulator. The CRT port `0x3d4` reads the characters to be printed from the CRT buffer frame which is at memory location `0xb8000` and displays the characters entered in the buffer.
- The function `uartputc` is responsible for the serial port that prints characters onto the Linux terminal.
- The `consputc` function uses both `cgaputc` and `uartputc` to ensure both QEMU and terminal display output are in sync.
- Using the cprintf function we observed that the left, right, up, down arrow key presses have character codes = 228, 229, 226 and 227 respectively

The following changes were made in order to implement caret navigation

Files modified:

1. **`console.c` :**
   - **`void cgaputc(int c):`**
     - In the BACKSPACE case, all characters to the right of current cursor position were shifted one place to the left in CRT buffer.
     - else if cases for LEFT_ARROW and RIGHT_ARROW which appropriately moves the cursor position using the pos variable.
     - In case the cursor is in the middle of a line and a new character is entered, we shift the CRT buffer one place to the right to make place for the new character. The variable `max_pos` keeps track of the rightmost position of the CRT buffer.

```c
else if (c == BACKSPACE)
{
  if (pos > 0)
  {
    --pos;
    for (int t = pos; t < max_pos; t++)
      crt[t] = ((crt[t + 1] & 0xff) | 0x0700);
    crt[max_pos] = ' ' | 0x0700;
    --max_pos;
  }
}

else if (c == LEFT_ARROW)
{
  --pos; // Move cursor to the left
}
else if (c == RIGHT_ARROW)
{
  ++pos; // Move cursor to the right
}
else
{
  for (int t = max_pos; t >= pos; t--)
    crt[t + 1] = ((crt[t] & 0xff) | 0x0700);
  max_pos++;
  if (pos > max_pos)
    max_pos = pos;
  crt[pos++] = (c & 0xff) | 0x0700; // black on white
}
```

   - A field `m` was added to the `input` structure to keep track of the rightmost of cursor in the input buffer.
   - **`void consputc(int c)`**

- RIGHT_ARROW and LEFT_ARROW cases were added. For uartputc, in the RIGHT_ARROW case, overwriting the character at the current cursor position makes the cursor move to the right. '\b' makes the cursor move to the left.
- In the BACKSPACE case, appropriate shifting of characters to the left takes place.
- Lastly, in case the cursor is in the middle of a line of text, the characters are moved to the right on the uart port to make space for the newly entered character.

```c
if (c == RIGHT_ARROW)
{
  uartputc(input.buf[input.e]);
  cgaputc(c);
  return;
}

if (c == LEFT_ARROW)
{
  uartputc('\b');
  cgaputc(c);
  return;
}
if (c == BACKSPACE)
{
  uartputc('\b');
  for (uint t = input.e; t < input.m; t++)
  {
    uartputc(input.buf[t + 1]);
    input.buf[t] = input.buf[t + 1];
  }
  uartputc(' ');
  uartputc('\b');
  for (uint t = input.e; t < input.m; t++)
  {
    uartputc('\b');
  }
}
```

```c
else
{
  if (input.e < input.m)
  {
    char fill, temp;
    fill = c;
    for (int i = input.e; i <= input.m; i++)
    {
      temp = input.buf[i % INPUT_BUF];
      uartputc(fill);
      fill = temp;
    }
    for (int i = input.e; i < input.m; i++)
    {
      uartputc('\b');
    }
  }
  else
  {
    uartputc(c);
  }
}

cgaputc(c);
```

Task 1.2) Shell History Ring

The following changes were made in order to implement caret navigation

Files modified:

1. **`console.c`:**
    - The historyRing structure was created for book keeping purposes. The description of the member variables is commented.

```c
struct
{
  char buffer[MAX_HISTORY][INPUT_BUF]; // 2D array to store commands
  int head;                            // Head of the queue
  int tail;                            // Tail of the queue
  int currentIndex;                    // Index of current command from history being viewed
  char partialBuffer[INPUT_BUF];       // Buffer to store command typed before accessing history
} historyRing;
```

- Steps involved for showing history commands on UP_ARROW and DOWN_ARROW are:
    - (Optional) Save partial command entered before accessing history
    - Clear the current line from display and input buffer.
    - Fill input buffer with history command from historyRing buffer.
    - Display the history command onto display
- The following functions were created to help implement the above steps:
    - **`void clearConsoleLine():`** Removes the line of text from the display. Since the cursor could be in between the line, it is first moved to the rightmost position using repeated **`consputc(RIGHT_ARROW)`** and then the entire line is deleted from display using repeated BACKSPACE.

- **`void clearInputBuffer()`**: This effectively clears the input buffer by resetting the **`input.e`** edit and **`input.m`** rightmost index to **`input.r`** which is the index of the first character of the line.
- **`void copyHistorytoInputBuffer()`**: This commands the command at index = **`historyRing.currentIndex`** in the **`historyRing`** buffer into the input buffer. It also appropriately changes the **`input.e`** and **`input.m`** values.
- **`void savePartialCommand()`**: This function is called when the user access the history commands for the first time, i.e when currentIndex = -1. It copies the command currently in input buffer into the historyRing.partialBuffer.
- **`void copyPartialToInputBuffer()`**: This function is called when the user presses DOWN_ARROW at the latest command stored in history. It copys the partial command saved before accessing history back into the input buffer.
- **`void saveHistory()`**: It is called when the user completes entering a command and before execution of the command. It performs the history buffer's circular queue head and tail pointer updates.

- In order to implement the history system call, the following function was created - int history(char *buffer, int historyID)
  - Since circular queue is used the given historyID is converted to an index wrt the head pointer.

```c
int history(char *buffer, int historyID)
{
  if (historyID < 0 || historyID > 15)
    return 2;
  int index = (historyRing.head + historyID) % MAX_HISTORY;
  if (historyRing.buffer[index][0] == 0)
    return 1;
  memmove(buffer, historyRing.buffer[index], INPUT_BUF);
  return 0;
}
```

2. **`history.c`**: This contains the user program for the history command. This will retrieve the latest 16 commands entered by the user using the history system call defined in user.h and prints the output onto the screen.

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

int main(int argc, char *argv[])
{
    char buffer[128];
    int hasHistory;
    for (int i = 0; i < 16; i++)
    {
        memset(buffer, 0, 128);
        if (history(buffer, i) != 0)
        {
            break;
        }
        hasHistory = 1;
        printf(1, "%d : %s\n", i, buffer);
    }
    if (hasHistory == 0)
    {
        printf(1, "No history available \n");
    }
    exit();
}
```

Output:

```
$ history
0 : ls
1 : echo hi
2 : wait2
3 : history
$ _
```

3. **`defs.h`**: The function prototype of the history function in **`console.c`** was added into this header file for the kernel system call **`sys_history`**.
4. **`sysproc.c`**: Implemented the kernel side system call **`sys_history`** function which first retrieves the arguments from user space and makes a call to the history function defined in **`console.c`**.
5. **`syscall.h`** : The **`SYS_history`** system call was assigned the number 22. This number defines the index of the system call in an array of function pointers in **`syscall.c`**
6. **`syscall.c`**: Linked externally defined **`sys_history`** function using extern **`int sys_history(void)`** and added a pointer to this function at index = 22 in the array of system call function pointers
7. **`user.h`**: Defined the function prototype **`int history(char*,int)`** for the user program called "history".
8. **`usys.S`**:
   1. Added the line **`SYSCALL(history)`**. Here **`SYSCALL`** is a macro that stores the index value of the system call defined in **`syscall.h`** in the **`%eax`** register. Using this index value, the corresponding system call is executed from **`syscall.c`**.
9. **`Makefile`** – The history user program defined in history.c was included in the list UPROGS so that it can be called as a command from the console.

Task 2) Statistics

Some observations made:

- **`proc.c`** contains process management related functions. Hence, our implementation of **`wait2`** was written in **`proc.c`**.
- The statistics must be updated in every clock cycle- called ticks in XV6. We traced the file responsible for updating ticks to be **`trap.c`**

Files modified:

1. **`proc.h`**
   - Extend the **`proc`** struct by adding the following fields to it:
     - **`ctime`** = creation time of process
     - **`stime`** = sleeping time of process
     - **`retime`** = ready time of process
     - **`rutime`** = running time of process
   - Also added function prototype for **`updateStats()`** function which is responsible for updating the above mentioned fields for each process.
2. **`proc.c`**
   - When a new process is created its state is **`UNUSED`**. **`allocproc`** function looks for process in **`UNUSED`** state and initialises them by changing their state to **`EMBRYO`**.
   - Hence,in **`allocproc`** we initialised **`ctime = ticks`**(current number of clock ticks) **`stime=0,retime=0, rutime=0;`**
   - Next we implemented **`updateStats()`** function. This function is run on every clock tick. This function is run atomically.

```
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);
}
```

- It iterates through every process in process table and checks their current state and increments the time spent in respective state accordingly. We do not update time in **`ZOMBIE`** state.

- The **`wait2`** function was implemented which is used by the kernel side system call **`SYS_wait2`** to retrieve the required statistics. It is an extension of the ordinary **`wait()`**.
  - First, we iterate through the process table to find a terminated child process.
  - If a child is found we return its PID and fill its retime etc in passed arguments.
  - Else if the current process itself was killed or has no child we return -1.

```c
int wait2(int *retime, int *rutime, int *stime)
{
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc();
  acquire(&ptable.lock);
  for(;;){
    // 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 pro
        *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;
      }
    }
```

3. `trap.c`
- To ensure **`updateStats`** runs on every clock tick, we declared its prototype in **`proc.h`** which is included in **`trap.c`**.
- **`trap.c`** is the file which deals with clock ticks and increments them. We call **`updateStats`** function there inside the **`trap`** function after every tick.

In order to implement the **`wait2`** system call, the following files were modified:

4. **`defs.h`**: Include prototype of **`wait2`** function for kernel side system call.
5. **`sysproc.c`** : Implement kernel side system call **`sys_wait2`** which makes a call to wait2 defined in proc.c
6. **`syscall.h`**: Assign the number 23 for **`SYS_wait2`** system call.
7. **`syscall.c`**: Add a pointer to the externally defined **`sys_wait2`** in the array of system calls at index = 23
8. **`usys.S`**: Add the line **`SYSCALL(wait2)`** which stores the value of the system call defined in syscall.h in some processor register. This will be used in syscall.c to find the index in the array of function pointers of system calls.
9. **`user.h`**: Add function prototype for the user side wait2 system call.
10. **`wait2.c`** : User level program to test wait2 system call.
    - It iterates twice with a **`fork()`** in each iteration.
    - **`wait2`** system call is then called and the returned value is displayed:
      - If wait2 returns -1: Indicates that no terminated child process was found
      - Else wait2 returns the pid of the terminated child along with ready, run and sleep times.

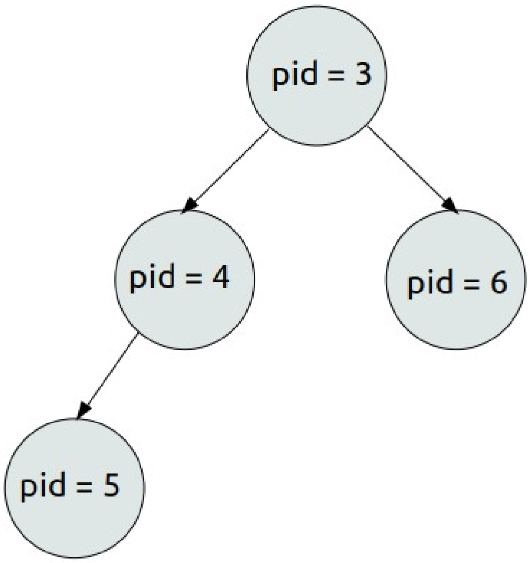
```c
int main()
{
    int i = 0;

    while (i < 2)
    {
        i++;
        int retime, rutime, stime;
        fork();
        int pid = wait2(&retime, &rutime, &stime);
        if (pid == -1)
        {
            printf(1, "No terminated child found for pid = %d\n", getpid());
            continue;
        }
        printf(1, "parent pid=%d, child pid:%d Retime:%d STime:%d Rutime:%d\n",
    }
    exit();
}
```

Output

```
No terminated child found for pid = 4
No terminated child found for pid = 5
parent pid=4, child pid:5 Retime:0 STime:0 Rutime:3
parent pid=3, child pid:4 Retime:0 STime:1 Rutime:9
No terminated child found for pid = 6
parent pid=3, child pid:6 Retime:0 STime:0 Rutime:3
```

Output Analysis:

We can decipher the parent child relations between the various processes from the output. The processes created can be described with the following process tree:



As expected, processes 5 and 6 have smaller runtimes compared to process 4 since these children are forked from their parents when i = 1 and do not have to run the loop.
Morever, when i = 1, process 4 spends some time in SLEEP state while waiting (using wait2) for process 5. Hence sleep time of process 4 is also non zero while that of process 5 and 6 is zero.