CS344 Assignment 2A

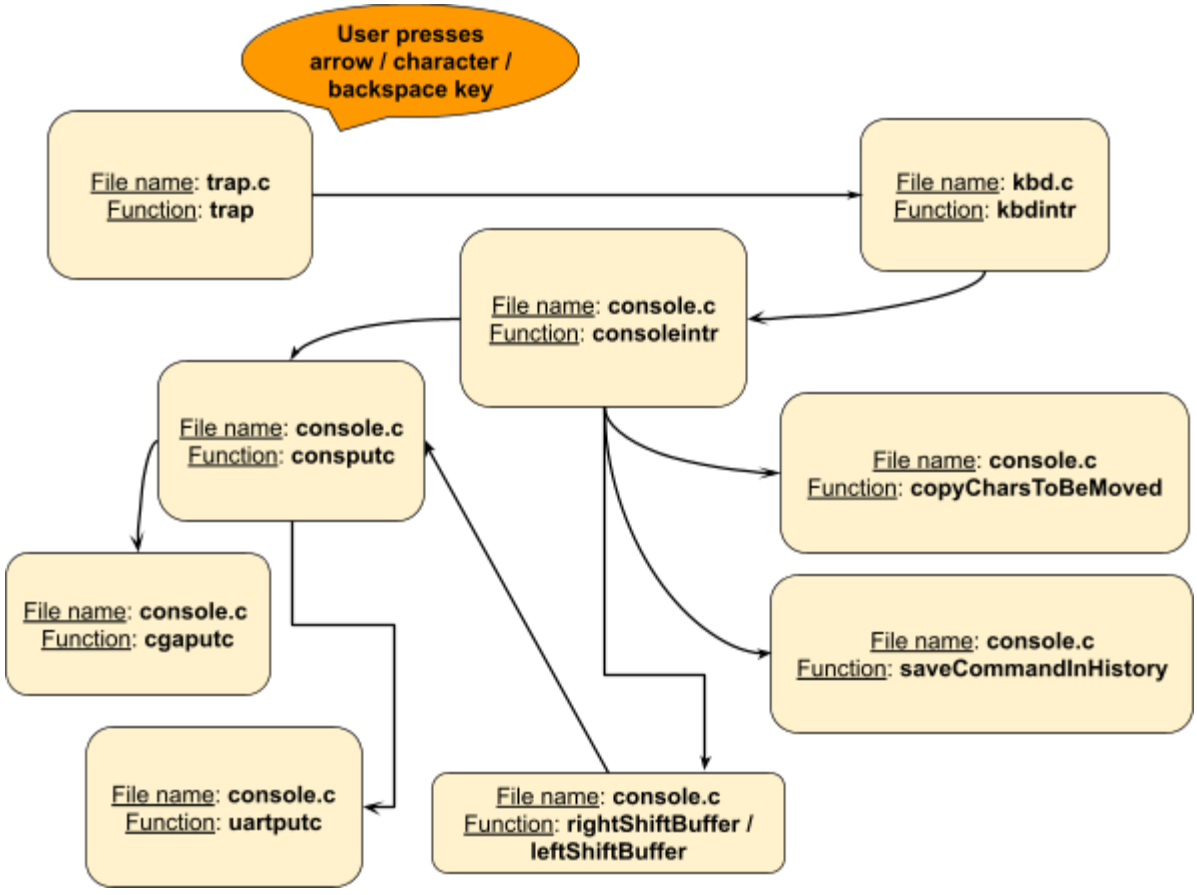
Group members:

- ★ Saket Kumar Singh 190101081
- ★ Pradnesh Prasad Kalkar 190101103
- ★ Mesharya M Choudhary 190101053
- ★ Parag Panigrahi 190101111

Task 1

1.1 Caret Navigation

Following **flowchart** describes the **sequence** of **functions** and **files** which are invoked when the user interacts with the qemu terminal window



Data Structures and functions that manage caret navigation

struct input (The parameters are described in the table)

```
130 struct {
131   char buf[INPUT_BUF];
132   uint r;  // Read index (first index of buf)
133   uint w;  // Write index (last index of buf)
134   uint e;  // Edit index (current index)
135   uint rightmost; // the first empty char in the line
136 } input;
```

Parameter Name	Description
buf	It is a character array which is used to store the value of the input.
r	This is the pointer to the first index of the buf.
w	This is the pointer to the last index of the buf.
e	This is the pointer to the current position of the carat.
rightmost	This is the pointer to the index after the last index of the buf.

copyCharsToBeMoved()

We **copy** the part of the **command** which is to be **shifted** left or right (depending upon whether a character is inserted or deleted and the position where this happens) **from** the **input buffer array to** the **charsToBeMoved array**.

```
/*
Store input.buf into charsToBeMoved (to use later)
Called when a new key is pressed and the cursor is not at EOL
*/
void copyCharsToBeMoved() {
  for (uint i = 0; i < (uint)(input.rightmost - input.r); i++) {
    charsToBeMoved[i] = input.buf[(input.e + i) % INPUT_BUF];
  }
}
```

rightShiftBuffer()

We **copy** the part of the command which was to be **shifted**, back to the **input Buffer array** from the **charsToBeMoved** array and we output the shifted characters onto the qemu terminal and the corresponding linux terminal. After this we reset the **charsToBeMoved array** for further use. This is followed by bringing the **caret** to it's appropriate position.

```
/*
Shift input.buf (backend) to the right by one unit and print the same on-screen (front-end)
Called when a new key is pressed and the cursor is not at EOL
*/
void rightShiftBuffer() {
  uint n = input.rightmost - input.e;
  for (uint i = 0; i < n; i++) {
    input.buf[(input.e + i) % INPUT_BUF] = charsToBeMoved[i];
    consputc(charsToBeMoved[i]);
  }

  // reset charsToBeMoved for further use
  for(uint i = 0; i < INPUT_BUF; i++) {
    charsToBeMoved[i] = '\0';
  }
  // return the caret to its correct position
  int i = n;
  while (i--) {
    consputc(LEFT_ARROW);
  }
}
```

leftShiftBuffer()

We **left shift** the appropriate part of the command in the **input buffer** and we **output** the shifted characters onto the qemu terminal and the corresponding linux terminal. Also we modify the **parameters** of the input buffer appropriately. This is followed by bringing the **caret** to its **appropriate position**.

```
/*
Shift input.buf (backend) to the left by one unit and print the same on-screen (front-end)
Called when a BACKSPACE is pressed and the cursor is not at EOL
*/
void leftShiftBuffer() {
  /**
   * For Ex: Input is abcdef and cursor is b/w c and d.
   * @cursor (display) : @pos in cgaputc
   */
  uint n = input.rightmost - input.e;
  consputc(LEFT_ARROW); // cursor (display) is b/w b and c
  input.e--; // set the backend part of cursor to the final correct position

  // abcdef -> abdeff
  for (uint i = 0; i < n; i++) {
    input.buf[(input.e + i) % INPUT_BUF] = input.buf[(input.e + i + 1) % INPUT_BUF];
    consputc(input.buf[(input.e + i + 1) % INPUT_BUF]);
  }
  // cursor (display) is b/w f and f

  input.rightmost--; // set input.rightmost to the final correct position
  consputc(' '); // delete the last char in line and advance cursor (display) by 1

  // set the cursor (display) to the correct position
  int i = n + 1;
  while (i--){
    consputc(LEFT_ARROW); // shift the caret back to the left
  }

  // at this point, the cursor (display) i.e. pos is in sync with input.e
}
```

eraseCurrentLineOnScreen()

Erases the **current line** from the **qemu** as well as the corresponding linux terminal.

```
/*
  Erase current line from screen
*/
void
eraseCurrentLineOnScreen(void) {
    int length = input.rightmost - input.r;
    while (length--) {
      consputc(BACKSPACE);
    }
}
```

copyCharsToBeMovedToOldBuffer()

This function **copies** the **command** we were typing before **accessing the history** from **input buffer** array to the **old buffer** array.

```
/*
  Copy input.buf into oldBuf
*/
void
copyCharsToBeMovedToOldBuffer(void) {
    oldBufferLength = input.rightmost - input.r;
    for (uint i = 0; i < oldBufferLength; i++) {
      oldBuf[i] = input.buf[(input.r + i) % INPUT_BUF];
    }
}
```

eraseContentOnInputBuffer()

This function **erases** the **content** present on the **input** buffer by resetting the corresponding parameters.

```
/*
  clear input buffer
*/
void
eraseContentOnInputBuffer(){
  input.rightmost = input.r;
  input.e = input.r;
}
```

copyBufferToScreen()

This function **prints** the **input character array** to the qemu as well as the corresponding linux terminal.

```
/*
  print bufToPrintOnScreen on-screen
*/
void
copyBufferToScreen(char* bufToPrintOnScreen, uint length){
  uint i = 0;
  while(length--) {
    consputc(bufToPrintOnScreen[i]);
    i++;
  }
}
```

copyBufferToInputBuffer()

This function **copies** the **input** character array to the input buffer array and **initializes** all the parameters corresponding to the input buffer appropriately.

```
/*
  Copy bufToSaveInInput to input.buf
  Set input.e and input.rightmost
  assumes input.r == input.w == input.rightmost == input.e
*/
void
copyBufferToInputBuffer(char * bufToSaveInInput, uint length){
  for (uint i = 0; i < length; i++) {
    input.buf[(input.r + i) % INPUT_BUF] = bufToSaveInInput[i];
  }

  input.e = input.r + length;
  input.rightmost = input.e;
}
```

consputc()

- consputc(int n) is used to **write characters** on consoles. This function uses two more functions, **cgaputc() writes the character to QEMU terminal** while **uartputc() prints the character to linux terminal.**
- By default we should directly print the character to the terminals, while handling some special characters like **BACKSPACE**, **LEFT_ARROW** and **RIGHT_ARROW**.
- For **BACKSPACE**, we are putting '\b'(backspace) characters at the cursor's position to move back, then placing an empty character and then again moving back using '\b'. Same kind of logic is used for the other special characters as well.
- For **LEFT_ARROW** we can simply move left by putting '\b'
- For **RIGHT_ARROW** if we are not at the end of the text, we can simply put whatever the text is present at the cursor's position back there and move forward. If it is at the end then do nothing.

```
switch (c) {
  case BACKSPACE:
    // uartputc prints to Linux's terminal
    uartputc('\b'); uartputc(' '); uartputc('\b');  // uart is writing to the linux shell
    break;
  case LEFT_ARROW:
    uartputc('\b');
    break;
  case RIGHT_ARROW:
    if (input.e < input.rightmost) {
      uartputc(input.buf[input.e % INPUT_BUF]);
      cgaputc(input.buf[input.e % INPUT_BUF], 1);
      input.e++;
    }
    else if (input.e == input.rightmost){
      consputc(' ');
      consputc(LEFT_ARROW);
    }
    break;
  default:
    uartputc(c);
}
if (c != RIGHT_ARROW) cgaputc(c, 0);
```

consoleintr()

- This handles the **interrupt** on typing any character.
- **BACKSPACE**: if the caret isn't at the end of the line, then we call the leftShiftBuffer() function to shift all the characters to the right of caret one space leftward and if the caret is at the end then we just delete the last character in the buffer.

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

- LEFT_ARROW: if we are not at the 0th index then we decrement edit index by one and call consputc() to show the changes. Same logic is used in RIGHT_ARROW too.

```
case LEFT_ARROW:
  if (input.e != input.w) {
    input.e--;
    consputc(c);
  }
  break;
case RIGHT_ARROW:
  consputc(RIGHT_ARROW);
  break;
```

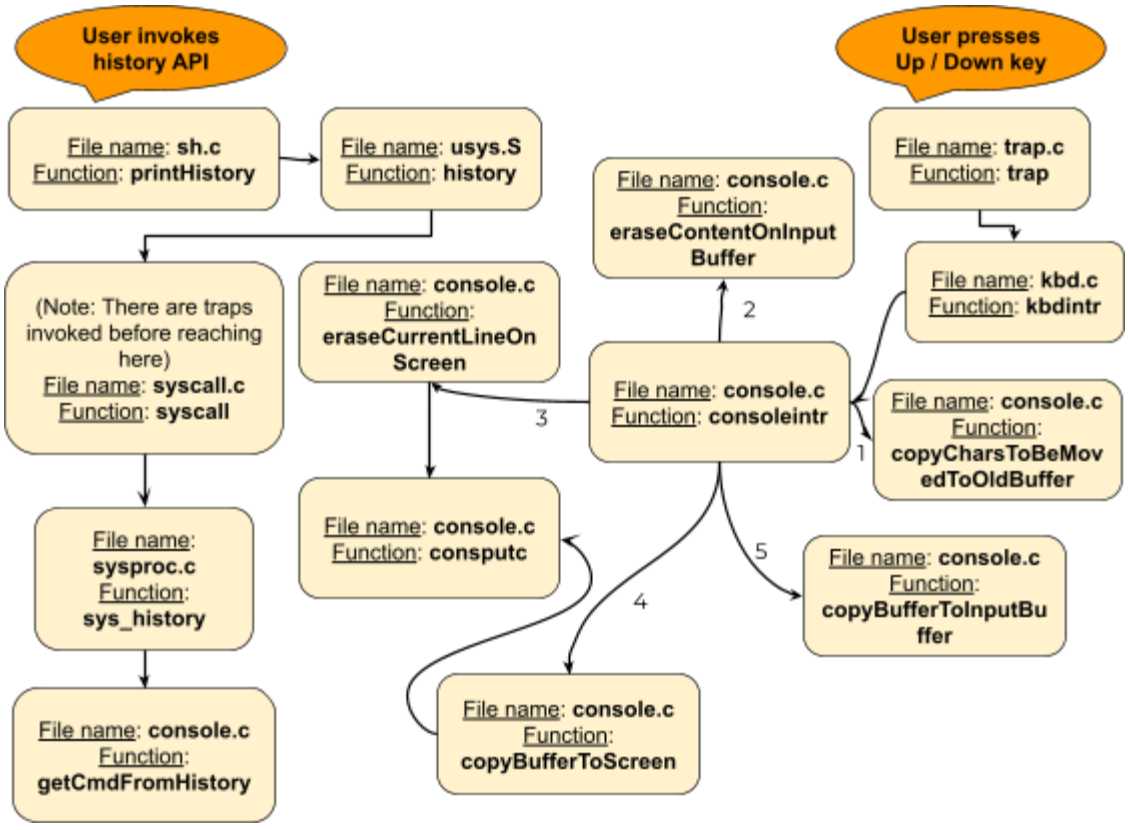
- '\n' or END_LINE:

When we press enter after bringing the caret in the middle of the command, the command executes normally. In addition to executing the command, we save whatever command is written in the buffer using the saveCommandInHistory(). This feature would be used in the shell history ring to get previous histories while using UP and DOWN arrow keys.

```
    }
    if(c == '\n' || c == C('D') || input.rightmost == input.r + INPUT_BUF){
      saveCommandInHistory();
      input.w = input.rightmost;
      wakeup(&input.r);
    }
  }
  break;
```

1.2 Shell History Ring

Following **flowchart** describes the **sequence** of **functions** and **files** which are invoked when the user interacts with the qemu terminal window -



historyBufferArray structure

This structure is used to store the **history of commands** used (maximum of **16**). We have stored the **oldest command** at position **15**. It comprises of 5 data structures:

- **bufferArr** : An array of strings of size MAX_HISTORY(16). It **stores** all the **commands**
- **lengthsArr**: An array of integers. **Stores length** of the **ith command** in bufferArr.
- **numOfCommmandsInMem**: Stores the **number** of **commands** we have typed
- **lastCommandIndex**: Stores the **index** of the **latest command** stored in the buffer.
- **currentHistory**: Helps us to find the **current history** we are **navigating** while using **UP_ARROW** and **DOWN_ARROW**. Stores the **offset** of **current history** from the **lastCommandIndex.**

```
struct {
  char bufferArr[MAX_HISTORY][INPUT_BUF]; // holds the actual command strings -
  uint lengthsArr[MAX_HISTORY]; // this will hold the length of each command string
  uint lastCommandIndex;  // the index of the last command entered to history
  int numOfCommmandsInMem; // number of history commands in mem
  int currentHistory; // holds the current history view -> displacement from the last command index
} historyBufferArray;
```

For **UP_ARROW**, we first check whether the **historyBufferArray.currentHistory** (an index in the historyBufferArray structure which helps us to find the command we are currently at in the history buffer) is valid or not. If valid we increment the index and erase the current text on screen using eraseCurrentLineOnScreen() function. If we are typing a new command before pressing this key then we store the currently written command on terminal in oldbuf array using copyCharsToBeMovedToOldBuffer() function in case we want to use it in the future . Then after clearing the input buffer we copy the command corresponding to the currentHistory into both the input buffer and to the screen.

```
case UP_ARROW:
 if (historyBufferArray.currentHistory < historyBufferArray.numOfCommmandsInMem-1 ){ // current history means the oldest possible will be MAX_HISTORY-1
    eraseCurrentLineOnScreen();

    // store the currently entered command (in the terminal) to the oldbuf
    if (historyBufferArray.currentHistory == -1)
        copyCharsToBeMovedToOldBuffer();

    eraseContentOnInputBuffer();
    historyBufferArray.currentHistory++;
    tempIndex = (historyBufferArray.lastCommandIndex + historyBufferArray.currentHistory) % MAX_HISTORY;
    copyBufferToScreen(historyBufferArray.bufferArr[ tempIndex]  , historyBufferArray.lengthsArr[tempIndex]);
    copyBufferToInputBuffer(historyBufferArray.bufferArr[ tempIndex]  , historyBufferArray.lengthsArr[tempIndex]);
 }
 break;
```

For **DOWN_ARROW**, the functions are similar to that above. Here if we haven't entered the history buffer yet (indicated by -1), we do nothing. If we are currently at the latest history command (indicated by 0) then we restore the contents in oldBuf. As mentioned above oldBuf would store whatever temporary command we wrote before pressing UP_ARROW key for the first time, so it would restore that command again. For example if we wrote drawtest (without running it), and then pressed UP_ARROW and then DOWN_ARROW it would restore drawtest again on the console. Also by default, the index would get decremented and would update the corresponding history on the screen as well as on the input buffer.

```
case DOWN_ARROW:
  switch(historyBufferArray.currentHistory){
    case -1:
      //does nothing
      break;

    case 0: // get string from old buf
      eraseCurrentLineOnScreen();
      copyBufferToInputBuffer(oldBuf, oldBufferLength);
      copyBufferToScreen(oldBuf, oldBufferLength);
      historyBufferArray.currentHistory--;
      break;

    default:
      eraseCurrentLineOnScreen();
      historyBufferArray.currentHistory--;
      tempIndex = (historyBufferArray.lastCommandIndex + historyBufferArray.currentHistory)%MAX_HISTORY;
      copyBufferToScreen(historyBufferArray.bufferArr[ tempIndex]  , historyBufferArray.lengthsArr[tempIndex]);
      copyBufferToInputBuffer(historyBufferArray.bufferArr[ tempIndex]  , historyBufferArray.lengthsArr[tempIndex]);
      break;
  }
  break;
```

For creating the history system call, we changed the following files

1. **user.h** - The function prototype for our history system call (for user-space) is added in this file. (declaration of history function).
2. **syscall.h** - The mapping from system call name (SYS_history) to system call number (23) is added in this file.
3. **syscall.c** - The mapping from system call number (23) to system call function(sys_history) is added in this file.
4. **usys.S** - The system call name is added in this file.
5. **sysproc.c** - The definition of the sys_history system call is added in this file

System call (sys_history)

```
93  /*
94    this is the actual function being called from syscall.c
95    @returns - 0 if suceeded, 1 if no history in the historyId given, 2 if illgal history id
96  */
97  int sys_history(void) {
98    char *buffer;
99    int historyId;
100   argptr(0, &buffer, 1);
101   argint(1, &historyId);
102   return getCmdFromHistory(buffer, historyId);
103 }
104
```

Here, the **sys_history**(void) **system call** extracts the arguments(buffer and historyId) from the history function declared in the user space (in user.h). Then it **calls** the function **getCmdFromHistory** which is declared in defs.h (for kernel-space).

Function getCmdFromHistory -

```
531 /*
532   this is the function that gets called by the sys_history and writes the requested command history in the buffer
533 */
534 int getCmdFromHistory(char *buffer, int historyId) {
535   // historyId != index of command in historyBufferArray.bufferArr
536   if (historyId < 0 || historyId > MAX_HISTORY - 1)
537     return 2;
538   if (historyId >= historyBufferArray.numOfCommmandsInMem) …
540   memset(buffer, '\0', INPUT_BUF);
541   int tempIndex = (historyBufferArray.lastCommandIndex + historyId) % MAX_HISTORY;
542   memmove(buffer, historyBufferArray.bufferArr[tempIndex], historyBufferArray.lengthsArr[tempIndex]);
543   return 0;
544 }
```

The function **getCmdFromHistory** contains the main code for the **history system call**. We have set the limit of the number of items in the history to be **MAX_HISTORY** = **16**, as directed in the assignment instructions. This function **fetches** the **history command** corresponding to the index historyId and **moves** it into the **buffer** which is passed as the first parameter to this function.

Function printHistory() - in file sh.c:

Inside the **main** function, there is a character array "**buf**" which stores the **current** command entered in the shell. In the below figure, we check if the command is equal to "**history**" followed by a new line since we go for execution of the history command. So, if the command is "history", **printHistory**() function is **called**.

```
187     if(buf[0] == 'h' && buf[1] == 'i' && buf[2] == 's' && buf[3] == 't'
188         && buf[4] == 'o' && buf[5] == 'r' && buf[6] == 'y' && buf[7] == '\n') {
189       printHistory();
190       continue;
191     }
```

Here below is the definition of the printHistory function in the file sh.c;

```
57  char cmdFromHistory[INPUT_BUF]; //this is the buffer that will get the current history command from history
58
59  /*
60    this function the calls to the different history indexes
61  */
62  void printHistory() {
63    int i, count = 0;
64    for (i = 0; i < MAX_HISTORY; i++) {
65      // 0 == newest command == historyId (always)
66      if (history(cmdFromHistory, MAX_HISTORY - i - 1) == 0) { // this is the sys call
67        count++;
68        if (count < 10)
69          printf(1, " %d: %s\n", count, cmdFromHistory);
70        else
71          printf(1, "%d: %s\n", count, cmdFromHistory);
72      }
73    }
74  }
```

The function printHistory() prints all the items stored in the history in order. The latest command we typed before corresponds to the historyId = 0, which will be the second parameter of the history system call. If we have 5 commands entered till now, then they will have the historyId from 0 to 4 with 4 as the oldest and 0 as the newest command. Inside the if condition, the history system call is invoked with the given parameters which in turn calls the sys_history() function which inturn calls getCmdFromHistory(). The command corresponding to the appropriate historyId is stored in the cmdFromHistory buffer when the history system call is invoked, and then we print it to the console using the file descriptor 1 in printf.

saveCommandInHistory()

This function is used to copy a command from the input buffer array to the history table at the appropriate position and then adjust the parameters for the table appropriately.

```
/*
  Copy current command in input.buf to historyBufferArray (saved history)
  @param length - length of command to be saved
*/
void
saveCommandInHistory(){
  uint len = input.rightmost - input.r - 1; // -1 to remove the last '\n' character
  if (len == 0) return; // to avoid blank commands to store in history

  historyBufferArray.currentHistory = -1; // reseting the users history current viewed

  if (historyBufferArray.numOfCommmandsInMem < MAX_HISTORY) {
    historyBufferArray.numOfCommmandsInMem++;
    // when we get to MAX_HISTORY commands in memory we keep on inserting to the array in a circular manner
  }
  historyBufferArray.lastCommandIndex = (historyBufferArray.lastCommandIndex - 1) % MAX_HISTORY;
  historyBufferArray.lengthsArr[historyBufferArray.lastCommandIndex] = len;

  // do not want to save in memory the last char '/n'
  for (uint i = 0; i < len; i++) {
    historyBufferArray.bufferArr[historyBufferArray.lastCommandIndex][i] =  input.buf[(input.r + i) % INPUT_BUF];
  }
}
```

Output for history command:

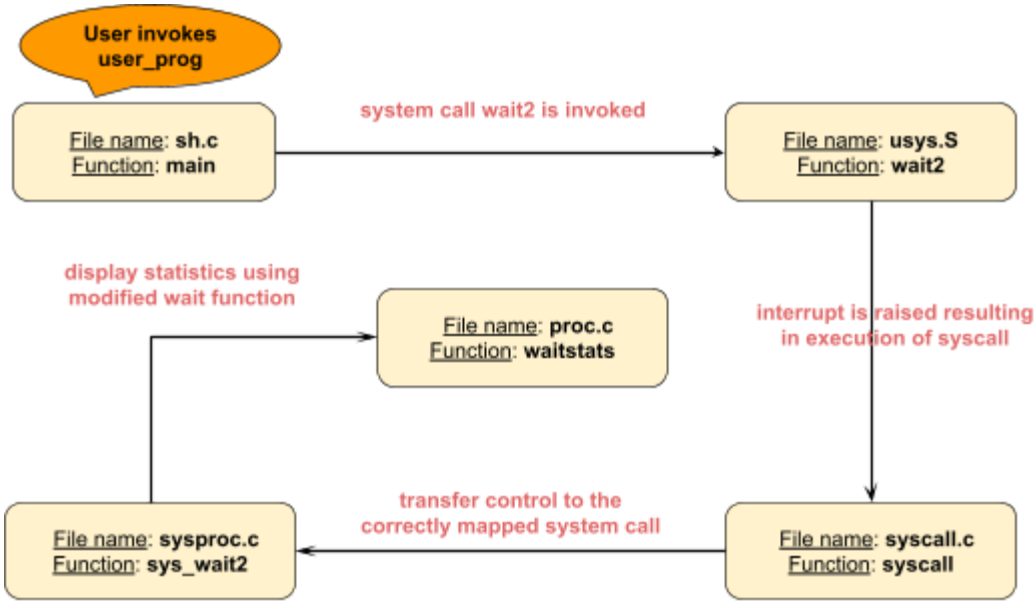
```
$ history
 1: ls
 2: abc
 3: history
$ f_
```

Task 2

Statistics

For creating the **wait2 system call**, we had to change these files -

1. **user.h** - The function prototype for our wait2 system call (for user-space) is added in this file. (declaration of wait2 function).
2. **syscall.h** - The mapping from system call name (SYS_wait2) to system call number (23) is added in this file.
3. **syscall.c** - The mapping from system call number (23) to system call function (sys_wait2) is added in this file.



4. **usys.S** - The system call name is added in this file.
5. **sysproc.c** - The definition of the sys_wait2 system call is added in this file.

[System Call (sys_wait2)]

```
29  /*
30    this is the actual function being called from syscall.c
31    @returns - pidof the terminated child process - if successful
32                -1, upon failure
33  */
34  int sys_wait2(void) {
35    int *retime, *rutime, *stime;
36    if (argptr(0, (void*)&retime, sizeof(retime)) < 0)
37      return -1;
38    if (argptr(1, (void*)&rutime, sizeof(retime)) < 0)
39      return -1;
40    if (argptr(2, (void*)&stime, sizeof(stime)) < 0)
41      return -1;
42    return waitstats(retime, rutime, stime);
43  }
```

Here, the function sys_wait2 is extracting the arguments from the wait2 user level function (API) which is declared in user.h; and storing them in **retime**, **rutime**, and **stime**. Afterwards, sys_wait2 calls the kernel-level function **waitstats** with the parameters retime, rutime and stime.

Function waitstats

```
73  static struct proc*
74  allocproc(void)
75  {
76    struct proc *p;
77    char *sp;
78
79    acquire(&ptable.lock);
80
81    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
82      if(p->state == UNUSED)
83        goto found;
84
85    release(&ptable.lock);
86    return 0;
87
88  found:
89    p->state = EMBRYO;
90    p->pid = nextpid++;
91    p->ctime = ticks;
92    p->retime = 0;
93    p->rutime = 0;
94    p->stime = 0;
```

When we create a new process, the process initialization happens in the **allocproc** function in proc.c as shown in the image above. On the lines 91, 92, 93 and 94 the values of ctime, retime, rutime and stime for the process are appropriately initialised.

The waitstats function implementation is shown in the image below -
Firstly we find the current process using the **myproc**() function. Then we place a lock on the process table since we are entering into a critical section as we access the process table and modify the entries. Then in an infinite for loop, we continuously look over all the processes currently in the process table and check if a process p is the child of curproc. If the p's state is **zombie**, i.e. it is

```
318  int waitstats(int *retime, int *rutime, int *stime) {
319    struct proc *p; // child process
320    int havekids, pid;
321    struct proc* curproc = myproc(); // parent process
322    acquire(&ptable.lock);
323    for(;;){
324      // find zombie children.
325      havekids = 0;
326      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
327        if(p->parent != curproc)
328          continue;
329        havekids = 1;
330        if(p->state == ZOMBIE) { // child is zombie
```

completed, we store the retime, rutime and stime of the process p into our waitstats function parameters, reset the values for the process p, release the lock and then return the pid of the child process. On line 358, the curproc waits for the child process to finish.
retime - The time spent by the process in the ready queue.
rutime - The time spent by the process running.
stime - The time spent by the process in the waiting queue.
pid - The unique identifier for a process.

Obtaining Process stats -

In sh.c, inside the main function, we focus on these lines of code. When we type the name of the user process, **fork1**() is executed with the shell process as the parent process and the user process **Userprog**.c as the child process. Parent process calls the **wait2** system call and this will return the **pid** of the Userprog user process (child process) and the retime, rutime and stime values will also be obtained. And then we just print the process statistics to the console.

```
168     if(fork1() == 0) {
169       // in child process (newly created)
170       runcmd(parsecmd(buf));
171     }
172     else {
173       // wait();
174       pid = wait2(&retime, &rutime, &stime); // system call
175       printf(1, "pid:%d retime:%d rutime:%d stime:%d\n", pid, retime, rutime, stime);
176     }
```

Inside trap.c, we are calling updatestats() function after every clock tick to update process statistics.

```
ticks++;
updatestats();
```

updatestats() - This function runs **every clock tick** and **updates** the **statistics fields**(retime, rutime, stime) for each of the **processes** that are **present** in the **process table**.

```
/*
  ass2:task2 this method will run every clock tick and update the statistic fields for each process
*/
void updatestats() {
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    switch(p->state) {
      case SLEEPING:
        p->stime++;
        break;
      case RUNNABLE:
        p->retime++;
        break;
      case RUNNING:
        p->rutime++;
        break;
      default:
        ;
    }
  }
  release(&ptable.lock);
}
```

Userprog.c - This is the **user program** which we execute and find the pid, retime, runtime, stime for and then print it.

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

int main(void)
{
    long long int sum = 0;

    for(long long i = 0; i < (long long) 1e9; i++){
        sum += i;
    }
    printf(1,"%d\n", sum);
    sleep(5);
    exit();
}
```

Output for the statistics test program (Userprog)

```
Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ Userprog
pid:3 retime:0 rutime:1 stime:5
$
```
