ASSIGNMENT 2A

CS344 OPERATING SYSTEMS LAB

Group 22

Shrey Verma 190101083

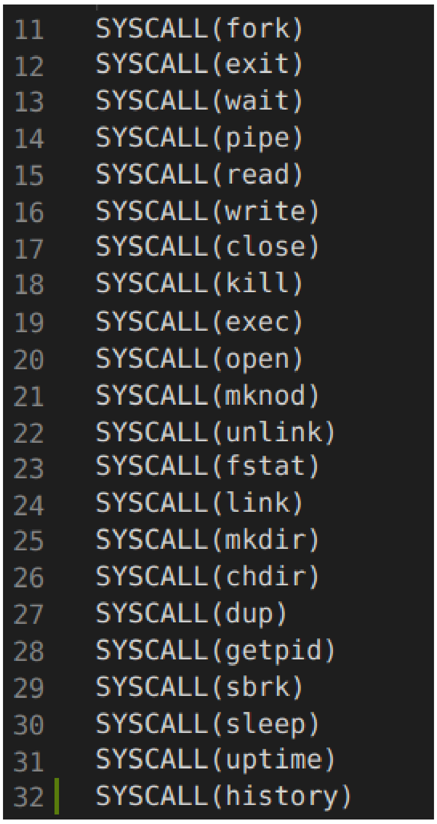
Rishav Mondal 190101072

Siddharth Charan 190101085

Karan Raj Sharma 190101043

Task 1:

USYS.S



```
SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(history)
```

This file contains the basic calls for system functions - these are the calls made from user mode. The Parameters given from the user are pushed to the stack and retrieved in the sys_*** implementation moving the "System call number" into eax and performing an interrupt.

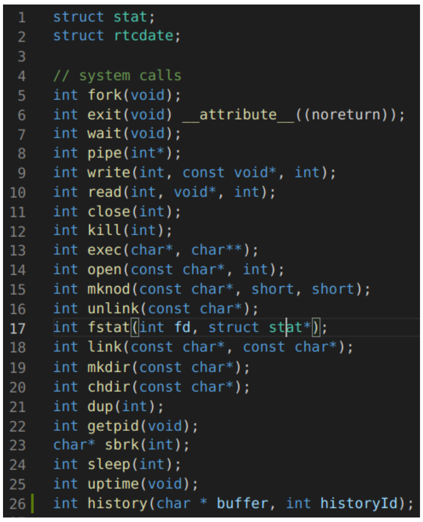
USER.H

This file contains the system call definitions in xv6.

Added the following line to this file: -

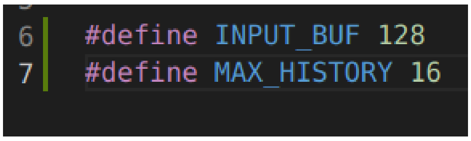
int history (char * buffer, int historyId);

This is the function that the user program will be calling. A call to the above function from a user program will be simply mapped to the system call number 22 which is defined as SYS_history preprocessor directive. The system knows what exactly is this system call and how to handle it.



```
struct stat;
struct rtcdate;

// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int history(char * buffer, int historyId);
```

TYPES.H



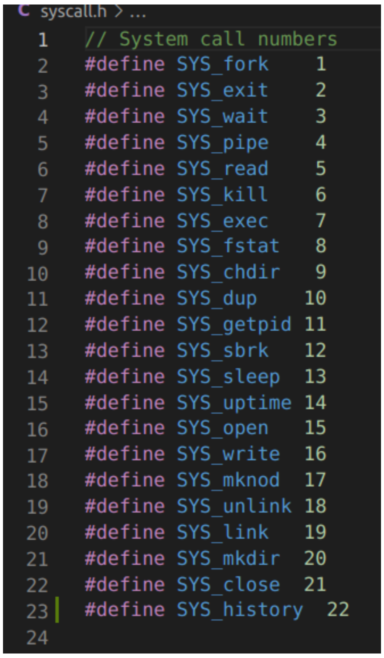
```
#define INPUT_BUF 128
#define MAX_HISTORY 16
```

The following constants are also defined using macros.

#define INPUT_BUF 128

#define MAX_HISTORY 16

SYSCALL.H



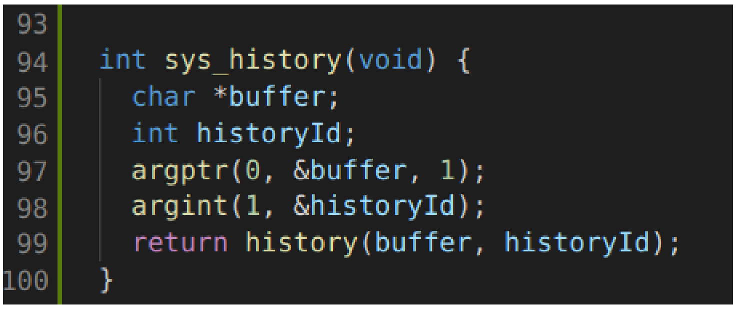
```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_history  22
```

Defined the index in the system call vector which gives our function a system call.

We have done this by adding the following line at the end of this file

#define SYS_history 22

SYSPROC.C



```

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

This file contains contains the implementations of process related system calls. We have implemented the the actual function being called from syscall.c in this file.

int sys_history(void) function returns - 0 if suceeded, 1 if no history in the historyId given, 2 if illgal history id

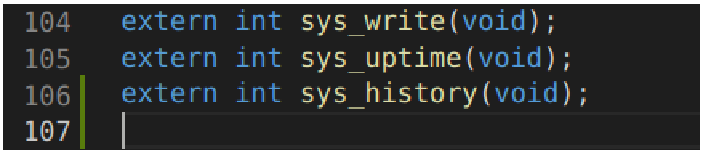
SYSFILE.C

Added the following line to this file: -

#include "console.h"

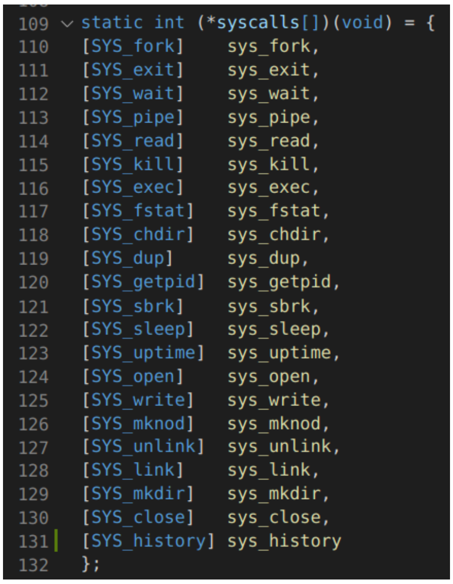
So that the newly created file console.h is included.

SYSCALL.C



```
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_history(void);

```

There's an array of function pointers inside this file with the function prototype static int *syscalls[])(void) .It uses the numbers of system calls defined above as indexes for a pointer to each system call function defined elsewhere. At the end of this function pointer array, we added the following line

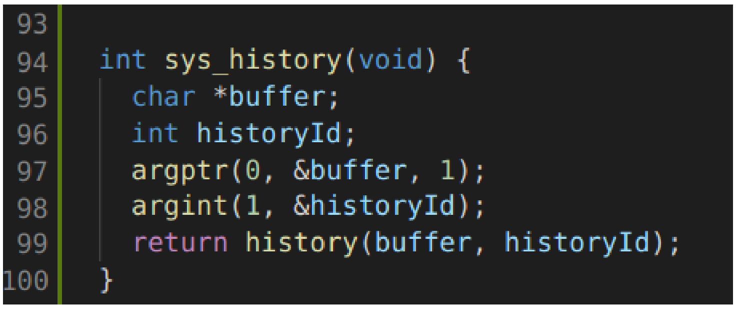


```
static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_history] sys_history
};
```

[SYS_history] sys_history,

Adding the line extern int sys_history(void); makes our function visible to the whole program. It connects the shell and the kernel. The system call function history was added to the system call vector at the position defined in syscall.h.

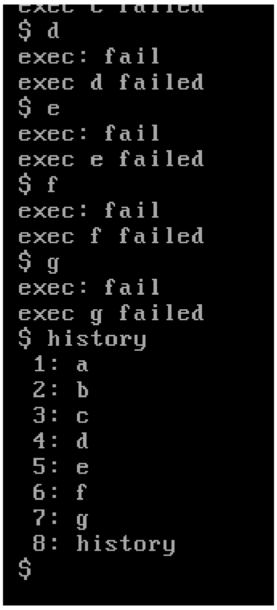
SH.C



```c

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

Void history1() This the function the calls to the different history indexes

cmdFromHistory : - this is the buffer that will get the current history command from history

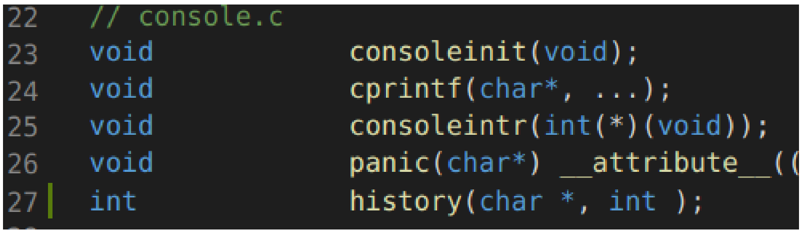


```
exec c failed
$ d
exec: fail
exec d failed
$ e
exec: fail
exec e failed
$ f
exec: fail
exec f failed
$ g
exec: fail
exec g failed
$ history
 1: a
 2: b
 3: c
 4: d
 5: e
 6: f
 7: g
 8: history
$
```

The history command was added to the shell user program so that it

upon writing the command a full list of the history should be printed to screen like in common.

DEFS.H



```c
// console.c
void            consoleinit(void);
void            cprintf(char*, ...);
void            consoleintr(int(*)(void));
void            panic(char*) __attribute__((
int             history(char *, int );
```

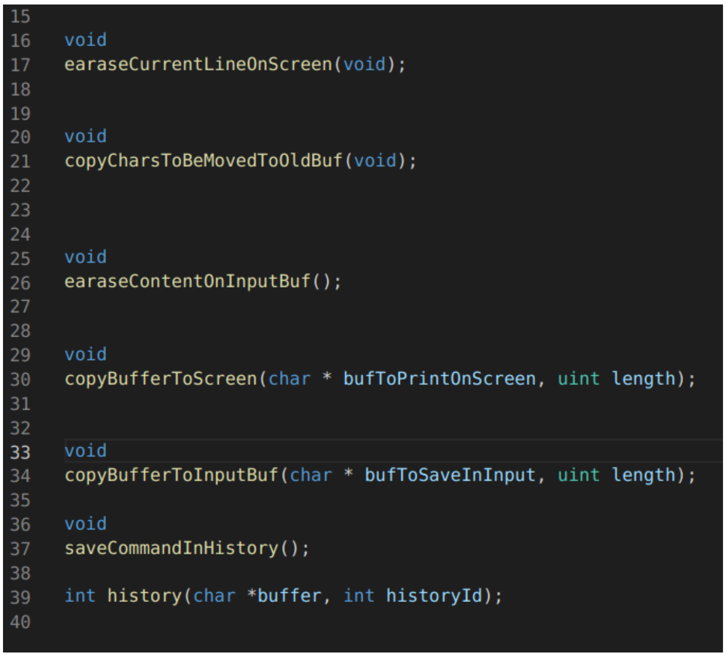
This file is used to add a forward declaration for your new system call

This line was added to this file: -

int history(char *, int);

CONSOLE.H

The definitions of functions inside console.c are prototyped in this file



```
void
earaseCurrentLineOnScreen(void);


void
copyCharsToBeMovedToOldBuf(void);



void
earaseContentOnInputBuf();


void
copyBufferToScreen(char * bufToPrintOnScreen, uint length);


void
copyBufferToInputBuf(char * bufToSaveInInput, uint length);

void
saveCommandInHistory();

int history(char *buffer, int historyId);
```

void earaseCurrentLineOnScreen(void); erases the current line from screen

void copyCharsToBeMovedToOldBuf(void); copies the chars currently on display (and on Input.buf) to oldBuf and save its length on current_history_viewed.lengthOld

void earaseContentOnInputBuf(); erases all the content of the current command on the inputbuf

void copyBufferToScreen(char * bufToPrintOnScreen, uint length);   this method will print the given buf on the screen

void copyBufferToInputBuf(char * bufToSaveInInput, uint length);   This function will copy the given buf to Input.buf and will set the input.e and input.rightmost

assumes input.r=input.w=input.rightmost=input.e

void saveCommandInHistory(); this function copies the current command in the input.buf to the saved history @param length - length of command to be saved

int history(char *buffer, int historyId); This function gets called by the sys_history and writes the requested command history in the buffer.

CONSOLE.C

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

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

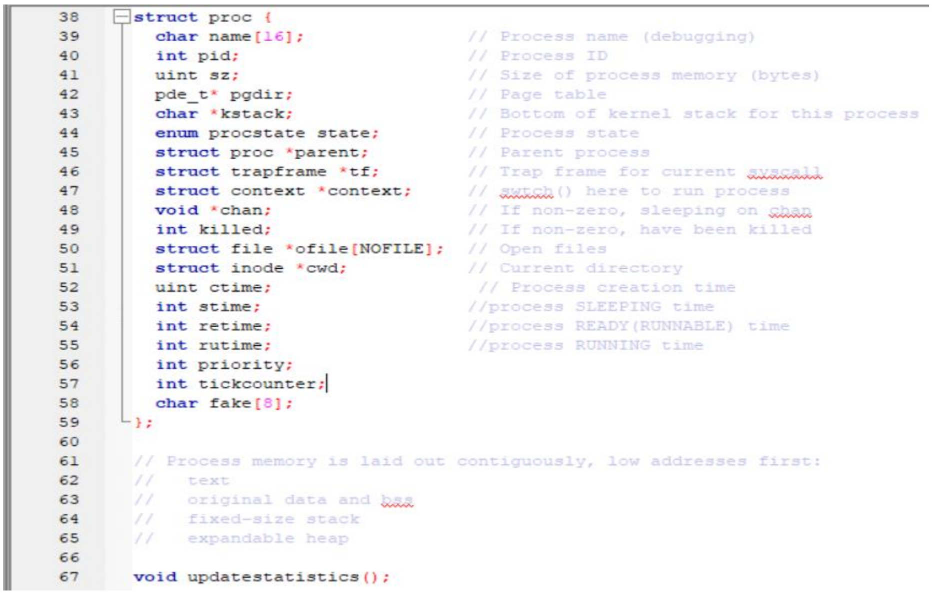
void shiftbufleft() : - Shift input.buf one byte to the left, and repaint the chars on-screen. Used Used only when punching in BACKSPACE and the caret isn't at the end of the line.

void copyCharsToBeMovedToOldBuf(void) : - this method copies the chars currently on display (and on Input.buf) to oldBuf and save its length on current_history_viewed.lengthOld

Task 2:

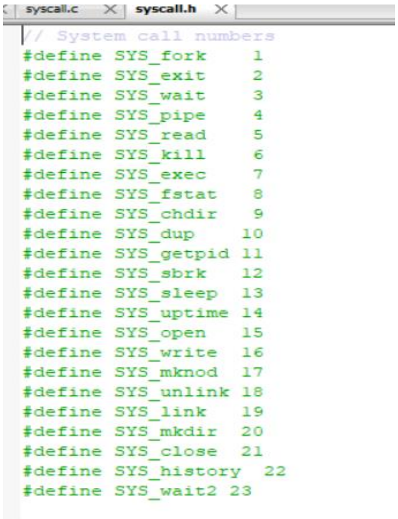
The following files were edited:

1. proc.h: the proc struct was extended by adding the following fields : ctime(process creation time), stime (sleeping time), retime (process ready time), rutime (process running time). Also, updatestatistics() function was declared.



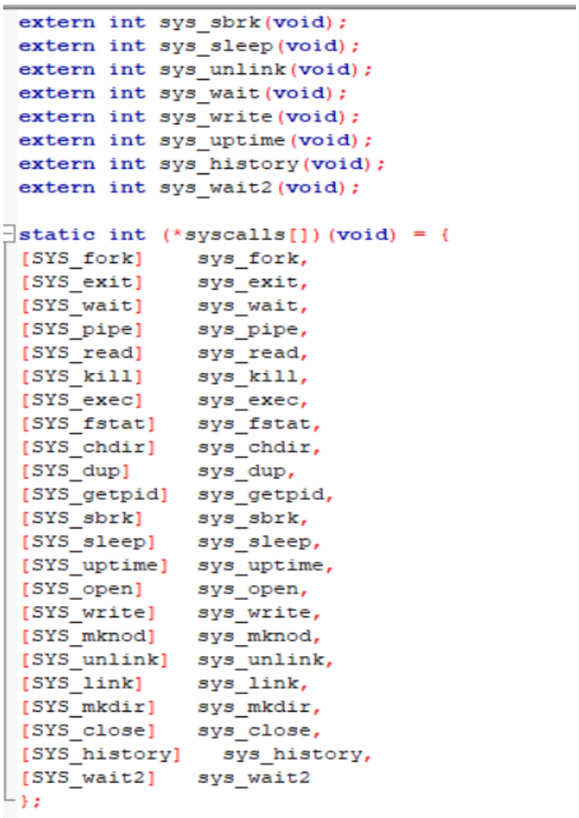
```
struct proc {
  char name[16];               // Process name (debugging)
  int pid;                     // Process ID
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  uint ctime;                   // Process creation time
  int stime;                   //process SLEEPING time
  int retime;                  //process READY(RUNNABLE) time
  int rutime;                  //process RUNNING time
  int priority;
  int tickcounter;
  char fake[8];
};

// Process memory is laid out contiguously, low addresses first:
//   text
//   original data and bss
//   fixed-size stack
//   expandable heap

void updatestatistics();
```

2. syscall.h: defined the index in the system call vector which gives our wait2 system call.



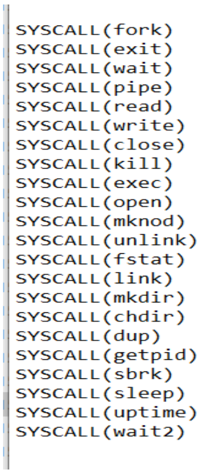
```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_history  22
#define SYS_wait2 23
```

3. syscall.c: extern int sys_wait2(void) makes our function visible to the whole program. It connects the shell and the kernel. The system call function wait2() was added to the system call vector at the position defined in syscall.h.



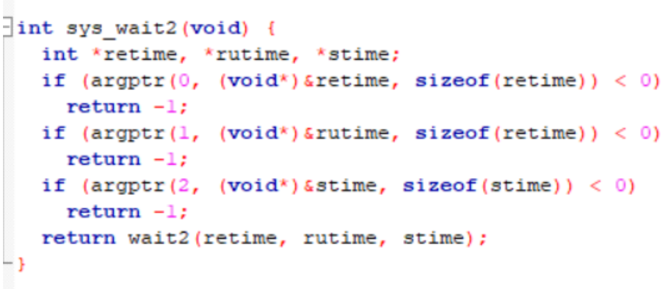
```
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_history(void);
extern int sys_wait2(void);

static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_history]   sys_history,
[SYS_wait2]   sys_wait2
};
```

4. usys.S: Creates a user level system call definition for the system call sys_wait2. Used this to connect the user's call to system function call.



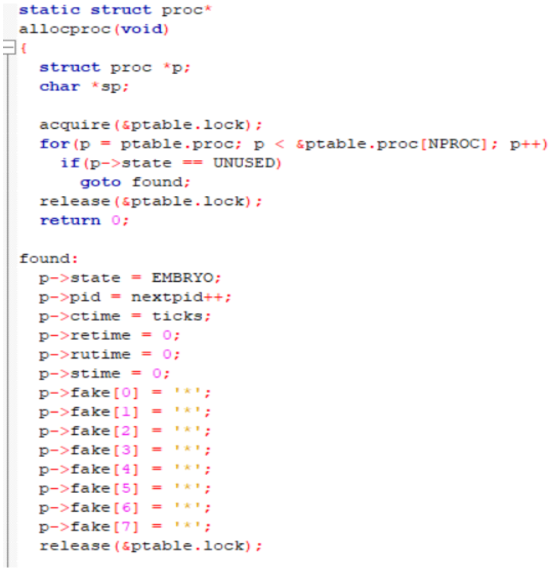
```
SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(wait2)
```

5. sysproc.c: created the function int sys_wait2(void) which calls wait2 function in case the call is valid, and returns -1 if not valid.

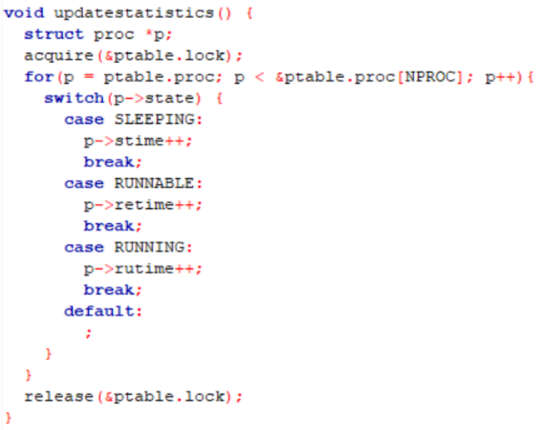


```
int sys_wait2(void) {
  int *retime, *rutime, *stime;
  if (argptr(0, (void*)&retime, sizeof(retime)) < 0)
    return -1;
  if (argptr(1, (void*)&rutime, sizeof(retime)) < 0)
    return -1;
  if (argptr(2, (void*)&stime, sizeof(stime)) < 0)
    return -1;
  return wait2(retime, rutime, stime);
}
```

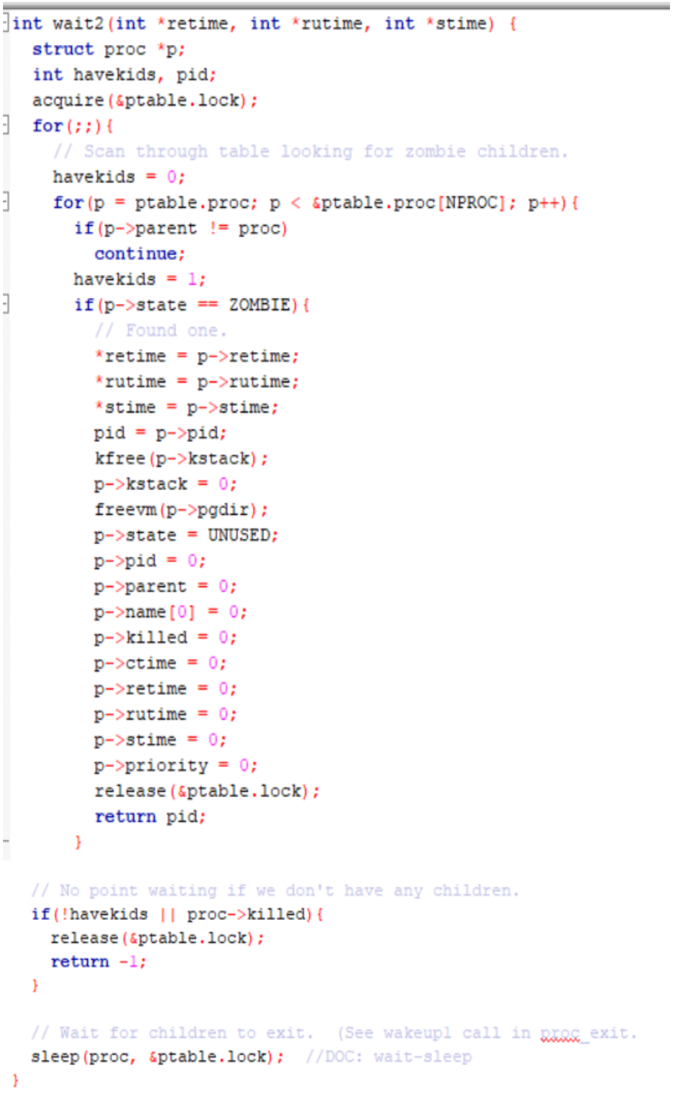
6. proc.c: created a new system call wait2() which extends the wait system call. It returns the pid of the terminated chid process if successful, and -1 upon failure. Also, updated the allocproc() function to initialize the new fields added in struct proc along with the old ones. Finally, created another function updatestatistics() which updates the stime, retime, rutime of the processes.



```
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;
  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->ctime = ticks;
  p->retime = 0;
  p->rutime = 0;
  p->stime = 0;
  p->fake[0] = '*';
  p->fake[1] = '*';
  p->fake[2] = '*';
  p->fake[3] = '*';
  p->fake[4] = '*';
  p->fake[5] = '*';
  p->fake[6] = '*';
  p->fake[7] = '*';
  release(&ptable.lock);
```

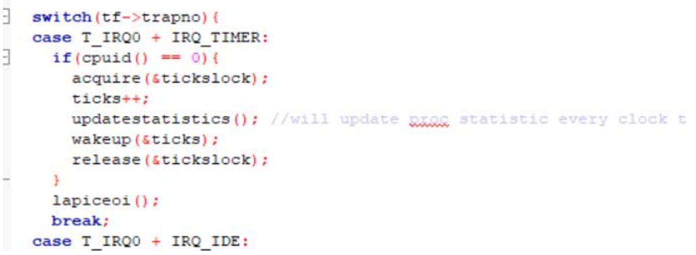


```
void updatestatistics() {
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    switch(p->state) {
      case SLEEPING:
        p->stime++;
        break;
      case RUNNABLE:
        p->retime++;
        break;
      case RUNNING:
        p->rutime++;
        break;
      default:
        ;
    }
  }
  release(&ptable.lock);
}
```



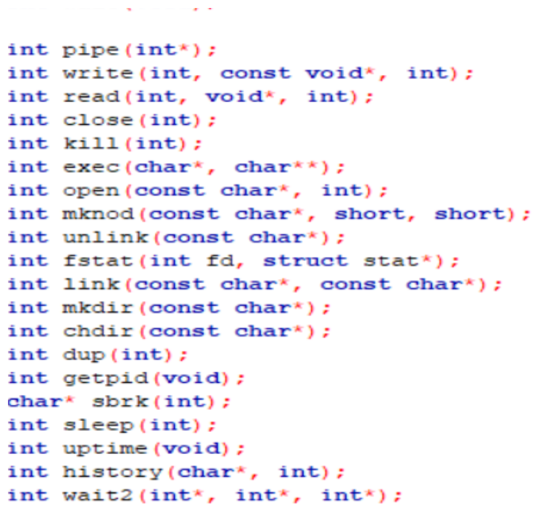
```
int wait2(int *retime, int *rutime, int *stime) {
  struct proc *p;
  int havekids, pid;
  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for zombie children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != proc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        *retime = p->retime;
        *rutime = p->rutime;
        *stime = p->stime;
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->state = UNUSED;
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->ctime = 0;
        p->retime = 0;
        p->rutime = 0;
        p->stime = 0;
        p->priority = 0;
        release(&ptable.lock);
        return pid;
      }

    // No point waiting if we don't have any children.
    if(!havekids || proc->killed){
      release(&ptable.lock);
      return -1;
    }

    // Wait for children to exit.  (See wakeup1 call in proc_exit.
    sleep(proc, &ptable.lock);  //DOC: wait-sleep
  }
```

7. trap.c: added the function call updatestatistics() inside the switch-case block which will update the process statistics every clock tick.



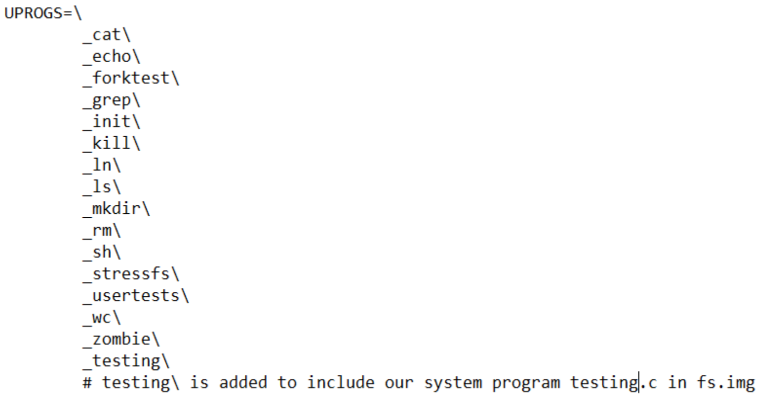
```
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      updatestatistics(); //will update proc statistic every clock t
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE:
```

8. user.h: added the declaration of the system call wait2().



```
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int history(char*, int);
int wait2(int*, int*, int*);
```

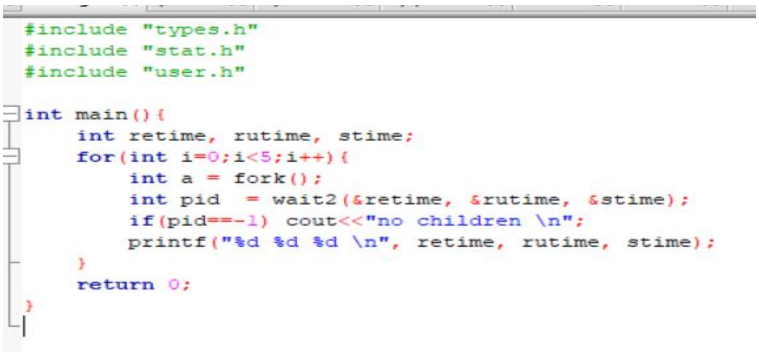
9. Makefile:



```
UPROGS=\
        _cat\
        _echo\
        _forktest\
        _grep\
        _init\
        _kill\
        _ln\
        _ls\
        _mkdir\
        _rm\
        _sh\
        _stressfs\
        _usertests\
        _wc\
        _zombie\
        _testing\
        # testing\ is added to include our system program testing.c in fs.img
```

In Makefile, we add our user program testing under the User Programs section (UPROGS). It is added to include it in fs.img

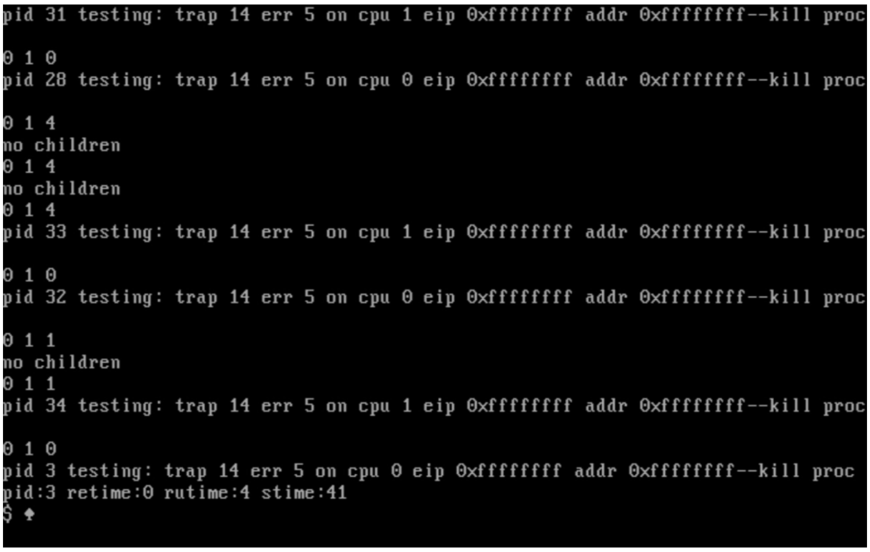
We created the user program testing.c to test the implementation of our modified code.



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

int main(){
       int retime, rutime, stime;
       for(int i=0;i<5;i++){
              int a = fork();
              int pid  = wait2(&retime, &rutime, &stime);
              if(pid==-1) cout<<"no children \n";
              printf("%d %d %d \n", retime, rutime, stime);
       }
       return 0;
}
```

Here, we are forking the current process multiple times and trying to get the updated ready time, running time, and sleeping time for every process. If there is no child of the process, we print so.

The output of the above code is:



```
pid 31 testing: trap 14 err 5 on cpu 1 eip 0xffffffff addr 0xffffffff--kill proc

0 1 0
pid 28 testing: trap 14 err 5 on cpu 0 eip 0xffffffff addr 0xffffffff--kill proc

0 1 4
no children
0 1 4
no children
0 1 4
pid 33 testing: trap 14 err 5 on cpu 1 eip 0xffffffff addr 0xffffffff--kill proc

0 1 0
pid 32 testing: trap 14 err 5 on cpu 0 eip 0xffffffff addr 0xffffffff--kill proc

0 1 1
no children
0 1 1
pid 34 testing: trap 14 err 5 on cpu 1 eip 0xffffffff addr 0xffffffff--kill proc

0 1 0
pid 3 testing: trap 14 err 5 on cpu 0 eip 0xffffffff addr 0xffffffff--kill proc
pid:3 retime:0 rutime:4 stime:41
$ ♠
```

The values of ready time, running time, and sleeping time get printed for every process. In case there is no child, the output is “no children”.