Shahriar Attar - 810100186 Sobhan Alaeddini - 810100188 Matin Bazrafshan - 810100093

Repository Link: https://github.com/Shahriar-0/Operating-System-Lab-Projects-F2024
Latest Commit Hash: 006d07550038ab2a31492c4983159e91613319d8

System Calls

1. Analyze `ULIB` libraries from the system calls point of view and explain them

In the xv6 operating system, `ULIB` is a library that contains the user-level implementations of system calls. System calls are the primary method by which user programs interact with the xv6 kernel. They provide an interface to the services made available by the operating system. `ULIB` variable consists of four objects:

```
ULIB = ulib.o usys.o printf.o umalloc.o
```

Makefile:146

overview of how system calls are implemented in xv6:

I. **Declaration**: System calls are declared in `syscall.h` and `user.h`. In `syscall.h`, each system call is assigned a unique number. In `user.h`, function prototypes for the system calls are declared[3].

II. **Implementation**: The actual implementation of system calls is done in `sysproc.c`. This is where the functionality of the system call is coded.

III. **Mapping**: In `syscall.c`, an array of function pointers called `syscalls` is used to map system call numbers to their respective implementations. This array uses designated initialization syntax, where the index corresponds to the system call number.

The `ULIB` library is used in xv6 for a few reasons:

- **Abstraction**: It provides an abstraction layer between the user program and the kernel. This means that user programs don't need to know the details of how system calls are implemented.
- **Safety**: It provides a safe interface for user programs to interact with the kernel. The library ensures that system calls are made correctly, preventing user programs from causing unintended effects in the kernel.

- **Convenience**: It simplifies the process of making system calls for user programs. Instead of manually setting up and making a system call, user programs can just call a function in the `ULIB` library.

Now we examine the system calls used in each of them.

- `ulib.o`: This file comes from `ulib.c`. Only two of the functions in this file make a system call:
  - `stat`: In this function, we use `open`, `fstat`, and `close`. `open` is used to open a file, `fstat` will read the file's metadata, and by `close` we show that we are done with that file.
  - `gets`: For reading from `stdin` we use `read` system call.
- `usys.o`: It's created using assembly code, At the beginning of the `usys.S` there is a macro that provides needed instructions for a system call:

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
```

usys.S:4

For instance, for a system call like fork, this code will run. The needed labels are defined in `syscall.h`.

```
.globl fork; \
  fork: \
    movl $SYS_fork, %eax; \ # syscall.h:2
    int $T_SYSCALL; \
    ret
```

The code works as below:

- First, we move the system call number for fork (defined as `SYS_fork` in `syscall.h`) into the `eax` register. The eax register is used to specify which system call to invoke.
- `int $T_SYSCALL`: This line triggers a software interrupt, which transfers control to the kernel. The kernel checks the `eax` register to determine which system call to perform. To perform appropriate system call we have this structure in `syscall.c` that will map the number to the function we want:

```c
#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
```

syscall.h:1

```c
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,
};
```

syscall.c:97

- `printf.c`: This file allows us to use `printf` in our code. We have three functions in this file, `putc` function uses `write` system call to write on `stdout`, and the other two functions, namely `printf` and `printint`, also call `putc` and therefore `write` system call.
- `umalloc.c`: In this file for `malloc` function we call `morecore` which calls `sbrk` to increase the process's memory when the process grows.
  All these functions' declarations are in `user.h`.

2. Ways to access kernel besides system calls in Linux

In Linux, besides system calls, there are a few other ways to interact with the kernel:

I. **Interrupts**: These are signals sent to the processor from hardware peripherals, like keyboards or network cards, to indicate that an event has occurred that requires attention. They are asynchronous and usually are received by I/O devices, for instance when we hit a keyboard key or move the mouse, or receive a packet through the network, we have an interrupt.

II. **Traps**: There are different types of traps:
   - System Calls
   - Exceptions: These are generated by the CPU when an error occurs, such as division by zero, invalid memory access, or an unknown instruction.
   - Signals: These are software interrupts that provide a method of handling asynchronous events. Some most used signals are `SIGINT`, `SIGKILL`, and `SIGTERM`.

III. **Kernel Modules**: These are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system.

IV. **`/proc` file system**: This is a pseudo-filesystem that provides an interface to kernel data structures. It can be used to obtain information about the system and to change certain kernel parameters at runtime (`sysctl`). `/dev` and `/sys` are other examples.

V. **`ioctl` system call**: This is a kind of device-specific system call. It provides a way to send complex commands to devices that cannot be expressed by regular system calls.

VI. **Shared Memory**: This is a method of interprocess communication (IPC) that allows multiple processes to share a memory segment.

VII. **Memory-mapped files**: This is a segment of virtual memory that has been assigned a direct byte-for-byte correlation with some portion of a file or file-like resource.

System Calls in xv6

3. Can other traps be triggered by `DPL_USER`

In the xv6 operating system, the `SETGATE` macro is used to set up the IDT[1] entries, including the system call trap gate. The `DPL_USER`[2] is set to allow user-mode programs to trigger the system call interrupt.

The DPL field in the gate descriptor determines the minimum privilege level a program must have to trigger the interrupt2. If the CPL[3] of the program is less than or equal to the DPL of the interrupt, the interrupt can be triggered.

If a user program tries to trigger another trap xv6 will block it with a `protection exception`. The reason for this is that if a malicious program could have triggered other traps, it would have had access to the kernel when it shouldn't have, and therefore it would have jeopardized the integrity of the system.

4. Reasons for pushing `ss` and `esp` on the kernel stack

First a quick review of, `SS` and `ESP`. They are registers in the x86 architecture that are related to the stack:

- `SS` (Stack Segment): This is a segment register in x86 architecture that points to the segment containing the stack. It is used in conjunction with the `ESP` register to keep track of the call stack, which is a critical data structure used by subroutines, interrupt handlers, and other programming constructs.
- `ESP` (Extended Stack Pointer): This is a register that points to the current top of the stack. It is automatically updated by push, pop, call, and ret instructions, as well as by interrupt and exception handling. The `ESP` register, together with the `SS` register, enables access to the stack segment, which is a region of memory used for dynamic storage of return addresses, local variables, and other functions or interrupt handler state.

In summary, we have two stacks user stack and kernel stack. When a mode change occurs from user mode to kernel mode, such as during a system call or interrupt, the CPU needs to save the context of the user mode process because now we want to use the kernel stack and we need `SS` and `ESP` to store values of kernel stack. By saving them, they can be restored when returning to user mode. If we don't change mode we don't need to change stack and therefore no need to push `SS` and `ESP`.

[1] Interrupt Descriptor Table
[2] User Descriptor Privilege Level
[3] Current Privilege Level

The reasons for having two stacks are as follows:

I. Preservation of Process State: The user mode stack contains information about the process's state before the mode switch. Saving `SS` and `ESP` allows the system to remember where the user mode stack is and what it contains, so it can be used again when the process returns to user mode.
II. Protection of User Mode Stack: The user mode stack may contain sensitive information. By switching to a separate kernel stack, the system can ensure that the kernel's operations do not accidentally overwrite or expose information on the user mode stack.
III. Prevention of Stack Overflows: Using a separate kernel stack helps prevent stack overflows. If the system used the user mode stack for kernel operations and the stack became too full, it could cause a stack overflow and potentially crash the system.
IV. Facilitation of Privilege Level Separation: Using separate stacks for user mode and kernel mode helps maintain the separation between different privilege levels. This is a key aspect of system security and stability.

high-level trap manager in C

5. How do have have access to function parameters in system calls

   `argint`, `argptr`, `argstr`, and `argfd` are functions used to retrieve arguments passed from user space to kernel space during a system call.

- `argint(int n, int *ip)`: This function retrieves the nth 32-bit integer argument of a system call. It uses pointer arithmetic to access the trapframe struct of the process, which contains the user-space registers of the syscall. The trapframe saved the function's parameters starting at the `esp` register. The function `fetchint` does some error checking and stores the value at the address specified by the `ip` pointer.
- `argptr(int n, char **pp, int size)`: This function retrieves the nth pointer argument of a system call. You need to give it the address of a pointer and the number of bytes of memory you want to fetch. Since a pointer in 32-bit architecture is 4-bytes, `argint` will also do the job. We check addresses are in the process's space so protection is met, because it may bug other processes' jobs.
- `argstr(int n, char **pp)`: This function retrieves the nth string argument of a system call. It uses the `argptr` function to fetch the argument as a pointer and then checks that this pointer points to a null-terminated string within the process's memory. If any function returns a negative value, that indicates an error, and the system call returns -1.
- `argfd(int n, int* pfd, struct file** pf)`: This function retrieves the nth file descriptor argument of a system call. You need to give it the address of an integer (to store the file descriptor) and the address of a pointer (to store the pointer to the file in the process's open file table). The function first calls `argint` to get the file descriptor as an integer. Then it checks if the file descriptor is valid (i.e., it is within the range of possible file descriptors and corresponds to an open file). If the file descriptor is valid, the function sets the integer and pointer to the file descriptor and the corresponding file.

An example of this parameter retrieves:

```
int sys_write(void) {
    struct file* f;
    int n;
    char* p;

    if (argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) <
0)
        return -1;
    return filewrite(f, p, n);
}
```

sysfile.c:75

`struct file* f` is a pointer to a `file` structure, which represents an open file in xv6. `char* p` is a pointer to the data to write. `int n` is the number of bytes to write.

`argfd(0, 0, &f)`: This retrieves the first argument of the system call, which is the file descriptor to write to. The file descriptor is an integer that the kernel uses to identify the open file. If this function returns a negative value, it indicates an error, such as an invalid file descriptor

.`argint(2, &n)`: This retrieves the third argument of the system call, which is the number of bytes to write.

`argptr(1, &p, n)`: This retrieves the second argument of the system call, which is a pointer to the data to write. The function checks that the pointer is valid and points to a region of memory that is at least `n` bytes in size.

`filewrite(f, p, n)`: If all the arguments are valid, the function calls `filewrite` to write the data to the file.

The `sys_write` function checks the return values of `argfd`, `argint`, and `argptr` to ensure that the arguments passed to the system call are valid. If any of these functions return a negative value, `sys_write` returns -1 to indicate an error. This could be due to reasons like an invalid file descriptor, an invalid pointer, or a size that is too large. By checking the arguments in this way, `sys_write` can prevent potential problems such as writing to an invalid location, writing too much data, or writing to a file that isn't open.

Checking System Call Steps in Kernel Using GDB

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

int main(int argc, char* argv[]) {
    int pid = getpid();
    printf(1, "Process ID: %d\n", pid);
    exit();
}
```

We wrote a simple program to get the `pid` of a program using `getpid()` system call.then we put a breakpoint at the beginning of the `syscall` function in `syscall.c` and continue until we hit it. Then we use `bt` or `where` to watch call stack.

GDB always traces and lists the chain of function calls that led to the current function. Command `bt` which is a short form of command `backtrace` shows the call stack of function calls leading to the current point of execution.

As you can see in the image below, `bt` shows a list of stack frames, each function has its own information which is local variables, arguments, return address, etc. The stack frame is organized in a last-in, first-out manner on the call stack, so we can backtrace our calls and find out where the bug occurs.

By calling `bt` we see this in output

```
syscall.c
>       struct proc* curproc = myproc();

        num = curproc->tf->eax;
        if (num > 0 && num < NELEM(syscalls) && syscalls[num]) {
            curproc->tf->eax = syscalls[num]();
        }
        else {
            cprintf("%d %s: unknown sys call %d\n",
                    curproc->pid, curproc->name, num);
            curproc->tf->eax = -1;
        }
    }

remote Thread 1.1 In: syscall                                   L123   PC: 0x80105630
(gdb) bt
#0  syscall () at syscall.c:123
#1  0x8010663d in trap (tf=0x8dffefb4) at trap.c:37
#2  0x801063ef in alltraps () at trapasm.S:20
#3  0x8dffefb4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
```

An overview of system call steps in xv6:

I. In `syscall.h` we assign a number for each system call
II. The declaration for it would be placed in `user.h`
III. The user program calls a system call function, which is a wrapper function defined in the user library. This function sets up the system call number and arguments and then triggers a software interrupt using the `int $T_SYSCALL` instruction. The code can be found in `usys.S`
IV. In `vectors.S` the `vector64` is already defined, by running the `int $T_SYSCALL` instruction we would enter here after pushing 64 (i.e. `$T_SYSCALL`) we would be transferred to `alltraps`.
V. The interrupt causes the CPU to switch from user mode to kernel mode and start executing the trap handler in the kernel. The trap handler is defined in `trapasm.S` and `trap.c`.
VI. The trap handler saves the state of the user program and checks the cause of the trap. If the cause is a system call, the handler calls the system call dispatcher in `syscall.c`.
VII. The system call dispatcher reads the system call number and maps it to that system call handler. It uses this number to index into a table of system call handler functions and calls the appropriate handler function.
VIII. The system call handler uses functions like `argint`, `argptr`, and `argstr` to fetch the system call arguments from the trap frame. These functions are also defined in `syscall.c`.
IX. The system call handler performs the requested operation and stores the return value in the trap frame.
X. Control returns to the trap handler, which restores the state of the user program and returns from the interrupt. This causes the CPU to switch back to user mode and continue executing the user program.
XI. The user program continues executing. If the system call function has a return value, the user program can use this value.

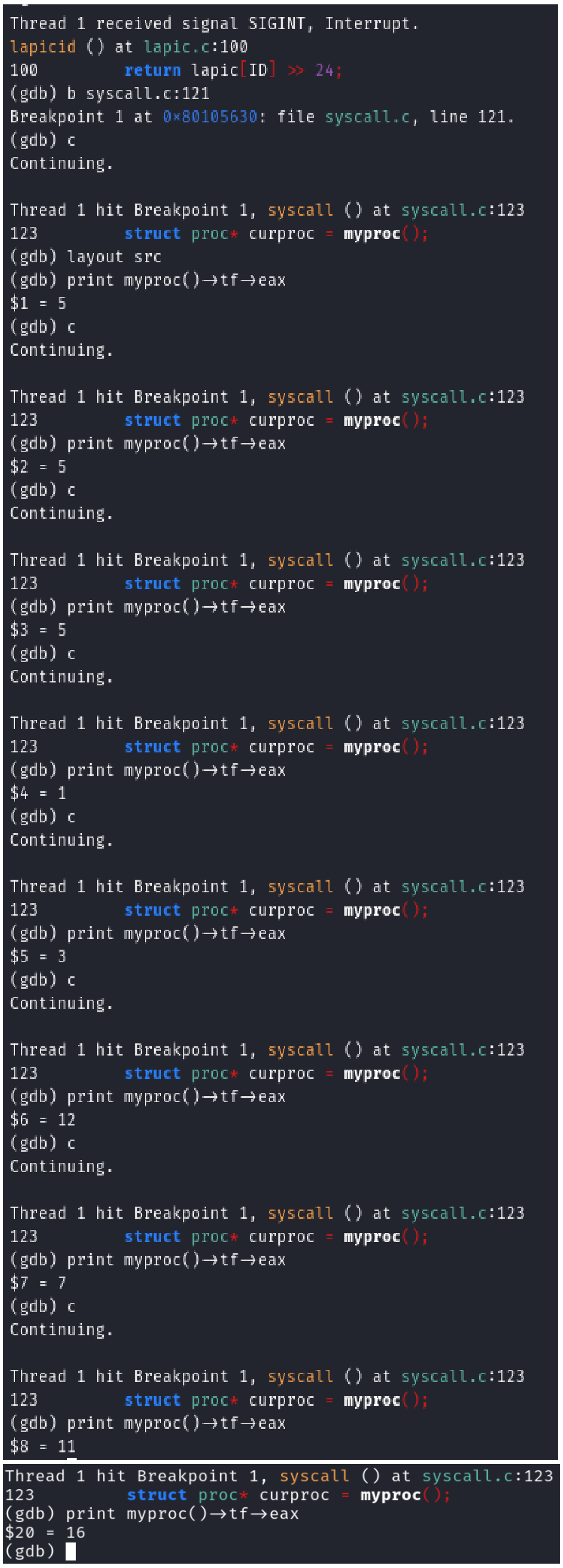
So the output of the call stack is justified.

To move between functions we can use these two commands

- command up is used to jump out of the function to where the function is called from. It has the same functionality as step out.
- command down is used to jump to the next function in the frame stack. It has the same functionality as step in.

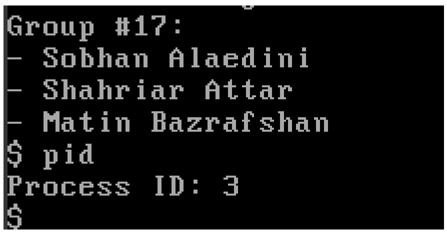
Because we are in the innermost frame we cannot go `down` anymore but by running `up` command we would go to the place where the `syscall` is called.

Then we check the EAX register value which holds the type of the system call



```
Thread 1 received signal SIGINT, Interrupt.
lapicid () at lapic.c:100
100           return lapic[ID] >> 24;
(gdb) b syscall.c:121
Breakpoint 1 at 0x80105630: file syscall.c, line 121.
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:123
123           struct proc* curproc = myproc();
(gdb) layout src
(gdb) print myproc()→tf→eax
$1 = 5
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:123
123           struct proc* curproc = myproc();
(gdb) print myproc()→tf→eax
$2 = 5
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:123
123           struct proc* curproc = myproc();
(gdb) print myproc()→tf→eax
$3 = 5
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:123
123           struct proc* curproc = myproc();
(gdb) print myproc()→tf→eax
$4 = 1
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:123
123           struct proc* curproc = myproc();
(gdb) print myproc()→tf→eax
$5 = 3
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:123
123           struct proc* curproc = myproc();
(gdb) print myproc()→tf→eax
$6 = 12
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:123
123           struct proc* curproc = myproc();
(gdb) print myproc()→tf→eax
$7 = 7
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:123
123           struct proc* curproc = myproc();
(gdb) print myproc()→tf→eax
$8 = 11
Thread 1 hit Breakpoint 1, syscall () at syscall.c:123
123           struct proc* curproc = myproc();
(gdb) print myproc()→tf→eax
$20 = 16
(gdb)
```

I. At first, we see multiple 5 which is the system call number for `read`, at first, it reads the input command from the terminal.
II. After that 1 is called which is for `fork`, we call it to create a new process for the user program.
III. Then 3 which is an indicator for `wait`, is shown, and we are waiting for our user program to run.
IV. Then by calling `sbrk` (number 12), we allocate memory for our process.
V. Then by calling the 7th system call which is `exec`, we run our `pid` program.
VI. After all of these, we see 11 which is what we were searching for, the `getpid` system call we used to obtain our process pid.
VII. In the end, we want to print the result so few `write` (which is shown by 16) are called and we see this in our terminal.



```
Group #17:
- Sobhan Alaedini
- Shahriar Attar
- Matin Bazrafshan
$ pid
Process ID: 3
$
```

Adding System Calls:

To add a new system call to the kernel, we need to commit these changes:

1. Add SYSCALL(<new system call>) to usys.S
    - usys.S is an assembly language file in xv6 that contains the user-side implementation of system calls.
    - SYSCALL is a constant defined in xv6 that represents the system call interrupt number.

```diff
4 xv6/usys.S
@@ -29,3 +29,7 @@ SYSCALL(getpid)
 SYSCALL(sbrk)
 SYSCALL(sleep)
 SYSCALL(uptime)
+SYSCALL(nuncle)
+SYSCALL(ptime)
+SYSCALL(fcopy)
+SYSCALL(droot)
```

2. Define a new system call in syscall.h
    - syscall.h is a header file that contains the definitions for system call numbers and function prototypes for system call implementations.

```diff
4 xv6/syscall.h
@@ -20,3 +20,7 @@
 #define SYS_link   19
 #define SYS_mkdir  20
 #define SYS_close  21
+#define SYS_nuncle 22
+#define SYS_ptime  23
+#define SYS_fcopy  24
+#define SYS_droot  25
```

3. Modify `syscall.c`, add new system call to `static int (*syscalls[])(void)` and `extern` it.
    - `static int (*syscalls[])(void)` is a declaration of a static array called syscalls. It is an array of function pointers, where each function pointer points to a system call implementation.
    - `extern` is used for forward-declaration.

```diff
8 xv6/syscall.c
@@ -93,6 +93,10 @@ extern int sys_unlink(void);
 extern int sys_wait(void);
 extern int sys_write(void);
 extern int sys_uptime(void);
+extern int sys_nuncle(void);
+extern int sys_ptime(void);
+extern int sys_fcopy(void);
+extern int sys_droot(void);
 
 static int (*syscalls[])(void) = {
     [SYS_fork] sys_fork,
@@ -116,6 +120,10 @@ static int (*syscalls[])(void) = {
     [SYS_link] sys_link,
     [SYS_mkdir] sys_mkdir,
     [SYS_close] sys_close,
+    [SYS_nuncle] sys_nuncle,
+    [SYS_ptime] sys_ptime,
+    [SYS_fcopy] sys_fcopy,
+    [SYS_droot] sys_droot,
 };
 
 void syscall(void) {
```

4. Add declaration of system calls to `defs.h`.

xv6/defs.h (4)

```diff
@@ -33,6 +33,7 @@ void fileinit(void);
 int fileread(struct file*, char*, int n);
 int filestat(struct file*, struct stat*);
 int filewrite(struct file*, char*, int n);
+int filecopy(struct inode* src, struct inode* dest);
 
 // fs.c
 void readsb(int dev, struct superblock* sb);
@@ -120,6 +121,9 @@ void userinit(void);
 int wait(void);
 void wakeup(void*);
 void yield(void);
+int nuncle(void);
+int ptime(void);
+int droot(int n);
 
 // swtch.S
 void swtch(struct context**, struct context*);
```

5. Add new system calls to `user.h`, to let user-space use these new system calls.

xv6/user.h (4)

```diff
@@ -23,6 +23,10 @@ int getpid(void);
 char* sbrk(int);
 int sleep(int);
 int uptime(void);
+int nuncle(void);
+int ptime(void);
+int fcopy(char*, char*);
+int droot(void);
 
 // ulib.c
 int stat(const char*, struct stat*);
```

6. And finally, we need to add the implementation of these system calls, to do this we separate the system call itself and its logic, and to do that we implement system calls in `sysproc.c` and `sysproc.h` with `sys_<systemcall>` naming format, inside of this function, we need to first, handle the inputs arguments and then call some other functions to do its logic.

I. fcopy - Copy a file into another file

First, we add `sys_fcopy` function in the file `sysfile.c`:

- We check if files exist.
- If the source file does not exist, we return -1.
- If the destination file does not exist, we create one, if it exists we warn the user that it will be overwritten.

This approach will result in the reusability of the `filecopy` function, so everywhere in the kernel, we can reuse this function. By having some logic implemented in `sys_fcopy`, we can use it even as a separate system call, so now we have a direct system call `fcopy`, and a reusable function `filecopy` for other parts of the kernel.

This is the implementation of the `sys_fcopy`:

```
int sys_fcopy(void) {
    char* src_path, *dest_path;
    if(argstr(0, &src_path) < 0 || argstr(1, &dest_path) < 0) {
        return -1;
    }

    begin_op();
    struct inode* src = namei(src_path);
    struct inode* dest = namei(dest_path);

    if(src == 0){
        cprintf("source file does not exist!\n");
        return -1;
    }

    if(src == dest) {
        cprintf("a file can not be copy at itself!\n");
        end_op();
        return -1;
    }

    else if(dest == 0) {
        dest = create(dest_path, T_FILE, 0, 0);
        iunlock(dest);
        if(dest == 0) {
            end_op();
            return -1;
        }
    }

    else {
        cprintf("[WARNING] destination file exists, it will be
overwriten!\n");
        end_op();
    }

    return filecopy(src, dest);
}
```

After that the process of copying a file is implemented inside `filecopy` function:

```
int filecopy(struct inode* src, struct inode* dest) {
    // copy buffer
    const int BUF_SIZE = 50;
    char buffer[BUF_SIZE];
    memset(buffer, 0, BUF_SIZE);

    // begin file system
    begin_op();

    // limit access to working files
    ilock(src);
    ilock(dest);

    // copy
    uint offset = 0;
    int bytes_read;
    dest->size = 0;
    while(1) {
        bytes_read = readi(src, buffer, offset, BUF_SIZE);
        if(bytes_read == 0)
            break;

        dest->size += bytes_read;
        writei(dest, buffer, offset, bytes_read);
        memset(buffer, 0, BUF_SIZE);
        iupdate(dest);
        offset += bytes_read;
    }

    // end
    end_op();

    // release files
    iunlockput(src);
    iunlockput(dest);


    return 0;
}
```

To test, we write a user-level, `fcopy_test,` program:

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

int main(int argc, char* argv[]) {
    if(argc != 3) {
        printf(2, "the command should be: fcopy_test <src> <dest>\n");
        exit();
    }
    fcopy(argv[1], argv[2]);

    exit();
}
```

The output looks like:

```
$ echo this is meant to be copied >> a.txt
$ cat a.txt
this is meant to be copied
$ fcopy_test a.txt a.txt
a file can not be copy at itself!
$ fcopy_test a.txt
the command should be: fcopy_test <src> <dest>
$ fcopy_test a.txt b.txt
$ cat b.txt
this is meant to be copied
$ echo this is another one >> c.txt
$ fcopy_test c.txt b.txt
[WARNING] destination file exists, it will be overwriten!
$ cat b.txt
this is another one
$
```

II. nuncle - Returns the number of uncle processes of the current process

- Add sys_nuncle to sysproc.c

```
// return number of uncles of a process
int sys_nuncle(void) {
    return nuncle();
}
```

- Add nuncle function to proc.c

```
int nuncle() {
    // get grandparent
    struct proc* current_proc = myproc();
    struct proc* grandparent = current_proc->parent->parent;

    // count all process' grandparent children, then minus 1
    int uncles = 0;
    struct proc* p;
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if(p->parent == grandparent)
            uncles++;
    }
    return uncles - 1;
}
```

Then we need to test it, we fork three processes with a process, then choose one of them, and then we fork another process, this new process is the grandchild of our first process and has two uncles. The implementation is in `nuncle_test.c`.

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

void test_nuncle(void) {
    int pid_c1 = fork();
    if(pid_c1 == 0){
        sleep(10);
        exit();
    }
    int pid_c2 = fork();
    if(pid_c2 == 0) {
        sleep(10);
        exit();
    }
    int pid_c3 = fork();
    if(pid_c3 == 0) {
        int pid_gc = fork();
        if(pid_gc == 0) {
            int n_uncle = nuncle();
            printf(1, "number of uncles: %d\n", n_uncle);
            exit();
        }
        wait();
        exit();
    }
    // wait for three children to exit
    wait();
    wait();
    wait();
}
```

And output looks like:

```
$ nuncle_test
number of uncles: 2
$
```

III. ptime – Returns processing time

To do this, we need to add a variable to hold the created time of a process. We add `uint ctime` to `struct proc` and assign its right value when a new process is created.

```diff
1 xv6/proc.h
@@ -54,6 +54,7 @@ struct proc {
   struct file* ofile[NOFILE]; // Open files
   struct inode* cwd;          // Current directory
   char name[16];              // Process name (debugging)
+  uint ctime;                 // created time
 };

```

now we initialize it:

- In xv6, `ticks` represent the number of timer interrupts that have occurred since the system was booted. The timer interrupt occurs 100 times per second so the gap between two ticks is equal to 10 milliseconds.

```diff
36 xv6/proc.c
@@ -207,7 +207,7 @@ int fork(void) {
     np->state = RUNNABLE;

     release(&ptable.lock);
-
+    np->ctime = ticks;
     return pid;
   }

```

Add `sys_ptime` function to `sysproc.c`:

```c
// return process time
int sys_ptime(void) {
    return ptime();
}
```

Add `ptime` function to `proc.c`:

```
int ptime() {
    struct proc* current_proc = myproc();
    return ticks - current_proc->ctime;
}
```

Then we write a test file. `ptime_test.c`:

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

void test_ptime(void) {
    int t;
    t = ptime();
    printf(1 ,"this process is created: %d milliseconds ago\n", 10 * t);
    sleep(100);
    t = ptime() - t;
    printf(1 ,"now it passed %d milliseconds again!\n",10 * t);
    int pid = fork();
    if(pid == 0) {
        sleep(100);
        int t_child = ptime();
        printf(1 ,"the child process lasts: %d milliseconds\n", 10 * t_child);
        exit();
    }
    else {
        wait();
        sleep(100);
        t = ptime();
        printf(1 ,"the father process lasts: %d milliseconds\n", 10 * t);
    }
}

int main(int argc, char* argv[]) {
    test_ptime();
    exit();
}
```

Result:

```
$ ptime_test
this process is created: 10 milliseconds ago
now it passed 1010 milliseconds again!
the child process lasts: 1000 milliseconds
the father process lasts: 3020 milliseconds
$
```

IV. droot - Returns the digital root of a given number

First we declare a `sys_droot` function in `sysproc.c`, this system call expects its argument to exist in its `ebx` register. So when we want to use this system call, instead of passing its argument to it(which behind the scenes will be pushed to a stack), we need to save them into the `EBX` register.

```
// return the digital root of number, read its argument from ebx
register
int sys_droot(void) {
    int n = myproc()->tf->ebx;
    return droot(n);
}
```

now we implement its logic inside `proc.c` file:

```
int droot(int n) {
    while(n > 9) {
        int sum_digits = 0;
        int temp = n;
        while(temp > 0) {
            sum_digits += temp % 10;
            temp /= 10;
        }
        n = sum_digits;
    }
    return n;
}
```

After that we need to write a test, so we create a `droot_test.c` file:

- The `volatile` keyword is used to prevent the compiler from optimizing or rearranging these assembly instructions.
- The value of `ebx` is saved into the local variable `prev_ebx` using the first `movl` instruction.
- The value of n (the command-line argument) is moved into the `ebx` register using the second `movl` instruction.
- After setting up the `ebx` register with the value of n, the program calls the `droot` function.
- After the `droot` function returns, the previous value of `ebx` (stored in `prev_ebx`) is restored using another inline assembly block.

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

int main(int argc, char* argv[]) {
    if(argc != 2) {
        printf(2, "the command should be: droot <n>\n");
        exit();
    }

    int n = atoi(argv[1]), prev_ebx;
    asm volatile (
        "movl %%ebx, %0;"
        "movl %1, %%ebx;"
        : "=r" (prev_ebx)
        : "r"(n)
    );
    int result = droot();
    asm volatile (
        "movl %0, %%ebx;"
        : : "r"(prev_ebx)
    );
    printf(1, "digital root of %d is %d\n", n, result);
    exit();
}
```

Now we can see the output:

```
$ droot_test 284
digital root of 284 is 5
$ droot_test 1234
digital root of 1234 is 1
$
```