Project Report: Implementation of System Call

Harrison Williams(861168815), Aadhar Chauhan(862546563)

Jan 25, 2025

Introduction

The goal of this project was to implement a custom system call within the xv6 operating system. The implementation involved defining a system call in both user and kernel space, processing user program parameters, and ensuring proper functionality through testing. This report outlines the steps undertaken to achieve the objectives. The YouTube video for our project can be found here: Project Demonstration Video

```
// 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_info   30  //lab1
```

```
extern uint64 sys_mkdir(void);
extern uint64 sys_close(void);
extern uint64 sys_info(void);  //lab1

// An array mapping syscall numbers f
// to the function that handles the s
static uint64 (*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_info]    sys_info,  //lab1
};
```

Figure 1: Defining the system call

Defining the System Call

(Aadhar Chauhan)
To define the system call, the following steps were executed:

1. **System Call Number:** A system call number was defined within the `syscall.h` header file (see Figure 1). This enables the kernel to identify the system call invoked by the user program from the system call table.

2. **Prototype Declaration:** In `syscall.c`, the function prototype of the new system call was declared using the `extern` keyword to indicate that the function is implemented elsewhere.

3. **Mapping System Call:** The system call number defined earlier was mapped to the function prototype in `syscall.c`. This mapping directs the kernel to the appropriate function when the system call is invoked.

Implementing the System Call

(Harrison Williams)

Kernel Implementation

1. **Process Struct Modification:** A new variable was added to the `struct proc` in `proc.h` to track the number of times the `info` system call is invoked by a process (see Figure 2).

```
// Per-process state
struct proc {
  struct spinlock lock;

  //lab1
  int infoCalls;              //track number of times info() is called
```

Figure 2: Changes in `proc.h` file

2. **Parameter Handling:** In `sysproc.c`, the `info` system call was implemented to retrieve an integer parameter from the user program using the `argint()` function. This parameter is passed to the kernel's implementation and increments the `infoCalls` variable (see Figure 3).

3. **Core Functionality:** The main implementation of the `info` system call was written in `proc.c` (see Figure 4). It processes the parameter passed and performs different tasks based on the parameter value, as described below:

    - **Case 1:** Counts the total number of processes in the system by iterating through the process array while holding a lock to avoid race conditions. (Aadhar Chauhan)

```c
//lab1- system call for proccess info
uint64
sys_info(void)
{
  int param; //user inputted parameter to pass to kernel

  //extract user parameter and pass to sys_call func. info in proc.c
  argint(0,&param);
  myproc()->infoCalls++; //increment var in struct proc to tack count

  return info(param);
}
```

```c
//lab1- system call info() implementation
int
info(int param)
{
  struct proc *p; //pointer to instance of struct proc
  int proc_count=0;

  //switch cases to process the various user inputted parameters
  switch(param){
```

Figure 3: Parameter Handling

- **Case 2:** Retrieves the `infoCalls` variable value, representing the number of `info` system calls made by the current process. (Aadhar Chauhan)
- **Case 3:** For parameter 3, the system call performs a page table walk to find the virtual addresses of pages stored in memory by the current process. An additional task was to count the number of pages above the address threshold of `0xF000000`. The design of the xv6 memory layout was analyzed to implement this functionality. xv6 uses 38-bit virtual addresses and a three-level page table hierarchy (`L2 → L0`) with 512 entries in each level. The traversal iterates through each level, indexing into entries where the page hit or valid page bit is set. xv6 provides a built-in function `walk()` that handles the page table traversal from `L2 → L0` for virtual addresses ranging from `0x0` to `MAXVA`, which is the maximum virtual address supported. During the traversal, valid pages are identified, and the total number of page hits and their virtual addresses are recorded. The virtual address of each hit is right-shifted by the corresponding level's lower bit range (`L2: 38-30, shift by 30 bits; L1: 29-21, shift by 21 bits; L0: 20-12, shift by 12 bits`). A bit-mask is then applied to extract the indices of the page table entries that caused the page hit at each level. (Harrison Williams)
- **Case 4:** Returns the virtual address of the kernel stack for the current process. (Harrison Williams)

User-Space Integration

(Aadhar Chauhan, Harrison Williams)

```
//switch cases to process the various user inputted parameters
switch(param){
  case 1: { //count #processes in system
    for(p=proc; p<&proc[NPROC]; p++){
      acquire(&p->lock);  //lock needed; avoid race conditions iter. proc array
        if(p->state != UNUSED) proc_count++; //count active proc only
      release(&p->lock);  //relase lock
    }
    return proc_count;
  }
  case 2: {//return count of # of info() func calls by cur_proc
    return myproc()->infoCalls;
  }
  case 3: {//#memory pages cur_proc stored in add. above 0xF000000
    uint64 numPages=0;  //total memory pages
    uint64 pagesAbove=0;//pages above 0xF000000
    struct proc *curproc = myproc(); //get current process
    pagetable_t pagetable = curproc->pagetable;
    uint64 threshold = 0xF000000; //address threshold

     printf("\n");
    //iterate through current process page table w/ xv6 walk() for each level (L2->L0)
    for (uint64 va = 0; va < MAXVA ; va += PGSIZE) { //range va 0 -> max VA xv6 allows
      pte_t *pte = walk(pagetable, va, 0);
      if (pte && (*pte & PTE_V)) {  //check page valid bit
        numPages++;

        //get page table indices from virtual address and perform bit shift & masking
        int l2_index = (va >> 30) & 0x1FF;  //L2 bits range (30-38)
        int l1_index = (va >> 21) & 0x1FF;  //L1 bits range (21-29)
        int l0_index = (va >> 12) & 0x1FF;  //L0 bits range (12-20)
        printf("Page found at indices: L2=%d, L1=%d, L0=%d (VA: 0x%lx)\n",
               l2_index, l1_index, l0_index, va);

        //check page va is above address threshold
        if (va > threshold) pagesAbove++;
      }
    }
    printf("Total Pages: %lu, Pages Above 0xF000000: %lu\n\n", numPages, pagesAbove);
    return pagesAbove;
  }
  case 4: {//address of kernel stack
    return myproc()->kstack;
  }
  default:
    return -1; //invalid paramter
}
```

Figure 4: Switch Implementation

1. **Stub Definitions:** The `usys.pl` file was updated to include system call stubs for the new function, allowing the trap handler to redirect the user program's call to kernel space with below line of code.

   ```
   entry("info");  #lab1 sys_call func info()
   ```

2. **Prototype Declaration:** The system call prototype was added to `user.h` to enable user programs to invoke it as follows.

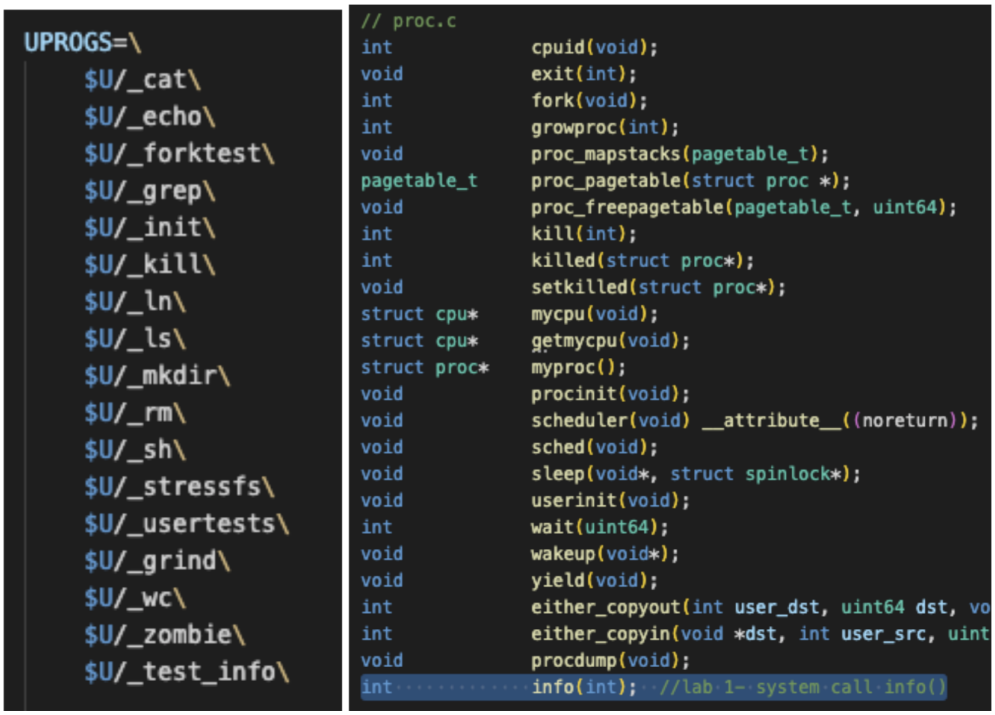
   ```
   int info(int);  #lab1 sys_call func info()
   ```

3. **Test Program:** A test program, `test_info.c`, was created in the user directory. This program tested the functionality of the implemented system call with various parameter values.

Compilation and Testing

(Aadhar Chauhan, Harrison Williams)

1. **Makefile Update:** The Makefile was modified to include the test program in the `UPROGS` field, enabling compilation as part of the xv6 user programs (see Figure 5).



```
UPROGS=\
    $U/_cat\
    $U/_echo\
    $U/_forktest\
    $U/_grep\
    $U/_init\
    $U/_kill\
    $U/_ln\
    $U/_ls\
    $U/_mkdir\
    $U/_rm\
    $U/_sh\
    $U/_stressfs\
    $U/_usertests\
    $U/_grind\
    $U/_wc\
    $U/_zombie\
    $U/_test_info\
```

```
// proc.c
int             cpuid(void);
void            exit(int);
int             fork(void);
int             growproc(int);
void            proc_mapstacks(pagetable_t);
pagetable_t     proc_pagetable(struct proc *);
void            proc_freepagetable(pagetable_t, uint64);
int             kill(int);
int             killed(struct proc*);
void            setkilled(struct proc*);
struct cpu*     mycpu(void);
struct cpu*     getmycpu(void);
struct proc*    myproc();
void            procinit(void);
void            scheduler(void) __attribute__((noreturn));
void            sched(void);
void            sleep(void*, struct spinlock*);
void            userinit(void);
int             wait(uint64);
void            wakeup(void*);
void            yield(void);
int             either_copyout(int user_dst, uint64 dst, vo
int             either_copyin(void *dst, int user_src, uint
void            procdump(void);
int             info(int);  //lab 1- system call info()
```

Figure 5: Compiling and Debugging

2. **Compiler Errors:** During compilation, an "implicit declaration" error was encountered. Investigation revealed that the `info` function needed to be declared in `defs.h`. Adding the declaration resolved the issue.

3. **Execution in QEMU:** The system was booted using QEMU, and the test program was executed. Outputs matched the expected results (see Figure 6), demonstrating the successful implementation of the system call. Specific results included:

   - Total process count.
   - Number of `info` system calls made.
   - Number of valid memory pages and addresses above a threshold.
   - Virtual address of the kernel stack.

Control Flow of the System Call Implementation

1. After xv6 is booted in QEMU, we call the user program executable `./test_info`.

2. A process runs the user program specified in the Makefile:

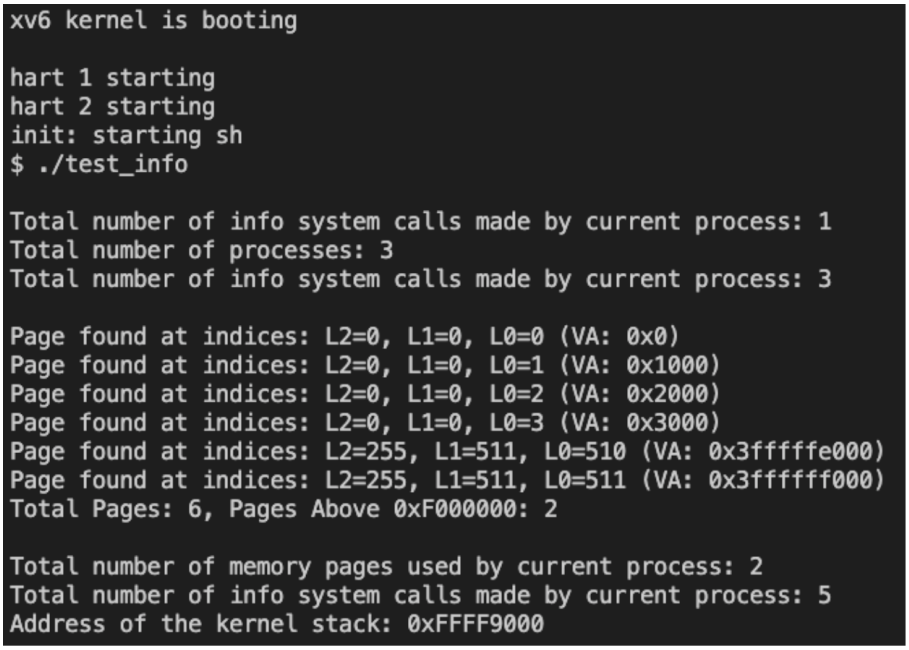


Figure 6: Our Results

```
UPROGS= $U/_test_info
```

3. The user program makes a series of system calls with user-specified parameters using the following function signature:

   ```
   int info(int parameter)
   ```

4. The system call prototype is defined in `user.h` as:

   ```
   int info(int);
   ```

   This allows the system calls to be invoked.

5. The system call stub, defined in the file `usys.pl`:

   ```
   entry("info")
   ```

   This passes the assembly code generated in `usys.s` to the trap handler, which bridges the gap between user space and kernel space.

6. The trap handler transfers control to the operating system in kernel mode to handle the trap. It checks the mapping of the system call made from user space.

7. Using the system call number defined in `syscall.h`:

```
#define SYS_info 30
```

This number serves as an index to access the array of system calls defined in `syscall.c`:

```
[SYS_info] sys_info
```

8. In `syscall.c`, the function prototype for the actual system call is declared:

```
extern uint64 sys_info(void);
```

This function is implemented in other source files.

9. The kernel executes the function `sys_info(void)` in `sysproc.c`. This function:
   (a) Takes the user parameter from the user program and passes it to the kernel using the xv6 function:

```
argint(int, *int)
```

   (b) Increments the added counter variable `infoCalls` in `struct proc` to track the number of system calls made by the process.

10. The `sys_info` function then calls the main system call to handle the user parameter passed from the user program. This jumps to `proc.c` to execute the main system call:

```
int info(int param)
```

which processes the output for the user parameter.

11. Before execution, it checks `defs.h` to verify the function prototype declaration:

```
int info(int);
```

12. The parameter enters a `switch` statement in `proc.c`, where a series of `case` statements provide the logic to return the correct output. (Details of this logic are explained in the previous section.)

13. Once the system call `info()` completes with the correct output for the user parameter, it stores this value in one of the kernel registers. This value is passed back to the user space and the user program, which continues running until the next trap or interrupt occurs for the OS to handle.

Conclusion

This project implemented a custom system call in xv6, covering kernel modifications, user-space integration, and testing. The system call performed various tasks based on user input, and the implementation was verified using QEMU. The experience demonstrated the complexities of system-level programming and reinforced knowledge of operating systems and xv6 architecture.

Video Demonstration

The project explanation and demonstration video can be accessed on YouTube using the following link:

Project Demonstration Video

Alternatively, here is the raw URL: https://www.youtube.com/watch?v=a7HDlhm28Bw

The github repo can be accessed via below link

Github_Project1