

Project Report on XV6 System Call Enhancement

Group 16

Xiao Fan, Kaya Gokalp, Ashwin Nellimuttath, and Hemanth Paladugu

October 20, 2023

Contents

1	Introduction	2
2	Video Link	2
3	List of All Files Modified	2
4	Detailed Explanation of Changes	2
4.1	Makefile	2
4.2	defs.h	3
4.3	kalloc.c	3
4.4	proc.c	4
4.5	syscall.c	5
4.6	syscall.h	6
4.7	sysproc.c	7
4.8	user.h	7
4.9	usys.pl	8
5	Output of the program	9
6	Description of XV6 Source Code Processing the ‘info’ System Call	9
7	Contributions of Each Member	9
8	Conclusion	10
9	References	10

1 Introduction

Operating systems serve as the intermediaries between hardware and software, governing how software applications access the underlying hardware. The XV6 operating system, a simplified version of UNIX v6, is an instructional tool offering insight into the internal workings of operating systems. In our project, we added a new system call named ‘info’ to XV6. This report chronicles our journey from conceptualizing the feature to its implementation.

2 Video Link

https://drive.google.com/file/d/143LRUQIqKHYc1YnhJTaIiLOzF1u3pvYj/view?usp=share_link

3 List of All Files Modified

To implement the ‘info’ system call, we had to modify several files, ensuring seamless integration and functionality. The files modified are:

- Makefile
- defs.h
- kalloc.c
- proc.c
- syscall.c
- syscall.h
- sysproc.c
- user.h
- usys.pl

4 Detailed Explanation of Changes

4.1 Makefile

The Makefile, responsible for compiling and linking the XV6 system, was updated to recognize our new user program, ‘sysinfotest’. This program tests the functionality of the ‘info’ system call.

▼ 1 Makefile		
	↑...	@@ -132,6 +132,7 @@ UPROGS=\
132	132	\$U/_grind\
133	133	\$U/_wc\
134	134	\$U/_zombie\
	135	+ \$U/_sysinfotest\
135	136	
136	137	fs.img: mkfs/mkfs README \$(UPROGS)
137	138	mkfs/mkfs fs.img README \$(UPROGS)

4.2 defs.h

In ‘defs.h’, we introduced new function prototypes related to the ‘info’ system call. This ensured that the system recognized our newly added functions.

▼ 5 kernel/defs.h		
	↑...	@@ -8,6 +8,7 @@ struct spinlock;
8	8	struct sleeplock;
9	9	struct stat;
10	10	struct superblock;
	11	+ struct pinfo;
11	12	
12	13	// bio.c
13	14	void binit(void);
	↑...	@@ -63,6 +64,7 @@ void ramdiskrw(struct buf*);
63	64	void* kalloc(void);
64	65	void kfree(void *);
65	66	void kinit(void);
	67	+ int num_free_pages(void);
66	68	
67	69	// log.c
68	70	void initlog(int, struct superblock*);
	↑...	@@ -106,6 +108,8 @@ void yield(void);
106	108	int either_copyout(int user_dst, uint64 dst, void *src, uint64 len);
107	109	int either_copyin(void *dst, int user_src, uint64 src, uint64 len);
108	110	void procdump(void);
	111	+ int total_active_process_count(void);
	112	+ void fill_pinfo(struct proc *curr_proc, struct pinfo *in);
109	113	
110	114	// swtch.S
111	115	void swtch(struct context*, struct context*);
	↑...	@@ -140,6 +144,7 @@ void argaddr(int, uint64 *);
140	144	int fetchstr(uint64, char*, int);
141	145	int fetchaddr(uint64, uint64*);
142	146	void syscall();
	147	+ int get_total_num_syscalls();

4.3 kalloc.c

‘kalloc.c’ required modifications to introduce a mechanism that computes free memory pages. This change was necessary for our ‘info’ system call to provide memory statistics.

```
kernel/kalloc.c
@@ -80,3 +80,24 @@ kalloc(void)
80      memset((char*)r, 5, PGSIZE); // fill with junk
81      return (void*)r;
82  }

83  +
84  + // Returns the number of free pages in the kmem.freelist
85  + int num_free_pages(void) {
86  +     int num_free_pages = 0;
87  +     struct run *r;
88  +
89  +     acquire(&kmem.lock);
90  +     r = kmem.freelist;
91  +
92  +     if (!r) {
93  +         return 0;
94  +     }
95  +
96  +     while (r) {
97  +         num_free_pages++;
98  +         r = r->next;
99  +     }
100  +     release(&kmem.lock);
101  +
102  +     return num_free_pages;
103  + }
```

4.4 proc.c

‘proc.c’ manages the processes in XV6. For the ‘info’ system call, we made several modifications:

- Introduced new data structures to capture and return process information.
- Added routines to compute the process’s memory usage and runtime.

These changes enable the ‘info’ system call to report vital process-specific statistics to user-level programs.

```

688 + int total_active_process_count(void) {
689 +     struct proc *p;
690 +     int total_num = 0;
691 +
692 +     for(p = proc; p < &proc[NPROC]; p++) {
693 +         acquire(&p->lock);
694 +         if(p->state == RUNNABLE ||
695 +            p->state == RUNNING ||
696 +            p->state == ZOMBIE) {
697 +
698 +             total_num++;
699 +         }
700 +
701 +         release(&p->lock);
702 +     }
703 +
704 +     return total_num;
705 + }
706 +
707 + // Fills pinfo struct based on the current process.
708 + void fill_pinfo(struct proc *curr_proc, struct pinfo *in) {
709 +     acquire(&curr_proc->lock);
710 +
711 +     int proc_used_bytes = curr_proc->sz;
712 +     int page_size = 4096;
713 +
714 +     // Efficient ivide with ceiling.
715 +     // Prone to overflow!!
716 +     int total_page_count = (proc_used_bytes + page_size - 1) / page_size;
717 +
718 +     in->ppid = curr_proc->parent->pid;
719 +     in->page_usage = total_page_count;
720 +     in->syscall_count = curr_proc->num_syscalls;
721 +     release(&curr_proc->lock);
722 + }

```

4.5 syscall.c

The ‘syscall.c’ file handles system call dispatching in XV6. To incorporate the ‘info’ system call:

- We incremented the system call count.
- Updated the system call function pointer table to include our new ‘info’ system call function.
- Made sure the correct system call number is associated with the ‘info’ function.

This ensures that the ‘info’ system call is dispatched correctly when invoked.

```

+ // Total number of syscalls done by the system so far.
+ static uint64 total_num_syscalls = 0;
+
+ int get_total_num_syscalls(void) {
+     return total_num_syscalls - 1;
+ }
+
+ // Prototypes for the functions that handle system calls.
+ extern uint64 sys_fork(void);
+ extern uint64 sys_exit(void);
+
+ extern uint64 sys_link(void);
+ extern uint64 sys_mkdir(void);
+ extern uint64 sys_close(void);
+ extern uint64 sys_sysinfo(void);
+ extern uint64 sys_procinfo(void);

```

```

+ [SYS_sysinfo] sys_sysinfo,
+ [SYS_procinfo] sys_procinfo
+ };
+
+ void
+ syscall(void)
+ {
+
+     num = p->trapframe->a7;
+     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
+
+ // Increase the number of syscalls done by this process.
+         p->num_syscalls++;
+ // Increase the number of syscalls so far done by the system.
+         total_num_syscalls++;
+
+ // Use num to lookup the system call function for num, call it,
+ // and store its return value in p->trapframe->a0
+         p->trapframe->a0 = syscalls[num]();
+
+     }
+ }

```

4.6 syscall.h

This header file in 'syscall.h' defines system call numbers. For our modifications:

- We appended new system call numbers for 'info'.
- Made sure there's no clash with existing system call numbers.

Ensuring that every system call has a unique number is vital for the kernel to distinguish between different system calls.

```

23 + #define SYS_sysinfo 22
24 + #define SYS_procinfo 23

```

4.7 sysproc.c

‘sysproc.c’ manages the processing of system calls at the user level. To accommodate our ‘info’ system call:

- We introduced new functions specific to the ‘info’ call.
- Ensured these functions can fetch the required system and process information and return them to the calling user program.

This guarantees that when a user program invokes the ‘info’ system call, it gets accurate and relevant data.

One critical thing to note here is the copyout operation for "sys-procinfo". After we fetch user pointer for pinfo struct to fill, we create our own pinfo and fill that. Since the created pinfo lives in kernel space we cannot directly move our pinfo to user located position. So we need to use copyout to copy from kernel space back to user space.

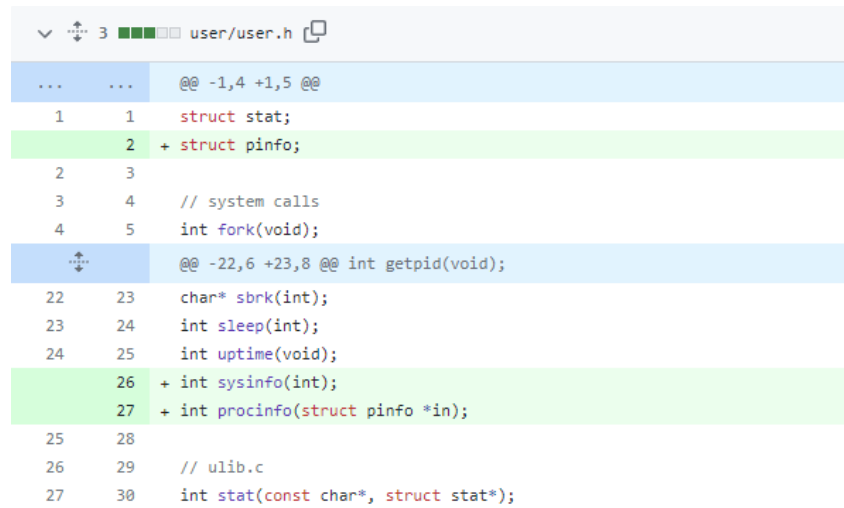
```
+
+ uint64
+ sys_sysinfo(void)
+ {
+     int param;
+     argint(0, &param);
+
+     if (param == 0) {
+         return total_active_process_count();
+     } else if (param == 1) {
+         return get_total_num_syscalls();
+     } else if (param == 2) {
+         return num_free_pages();
+     }
+
+     return -1;
+ }
+
+ uint64
+ sys_procinfo(void)
+ {
+     uint64 pinfo_ptr; // user pointer to pinfo struct to be filled.
+     argaddr(0, &pinfo_ptr);
+
+     if (pinfo_ptr == 0) {
+         // User provided nullptr.
+         return -1;
+     }
+
+     struct pinfo in;
+     struct proc *p = myproc();
+
+     // fills the pinfo information from current process.
+     fill_pinfo(p, &in);
+
+     // copy the pinfo struct back to user space.
+     if (copyout(p->pagetable, pinfo_ptr, (char *)&in, sizeof(in)) < 0)
+         return -1;
+     return 0;
+ }
```

4.8 user.h

In ‘user.h’, which serves as a header file for user-level interfaces:

- We introduced prototypes for the ‘info’ system call, ensuring that user programs can recognize and utilize the new system call.

This bridges the gap between our kernel modifications and user-level applications, allowing seamless interaction between the two.



```

...  ...  @@ -1,4 +1,5 @@
1    1    struct stat;
2    2    + struct pinfo;
2    3
3    4    // system calls
4    5    int fork(void);
@@ -22,6 +23,8 @@ int getpid(void);
22   23   char* sbrk(int);
23   24   int sleep(int);
24   25   int uptime(void);
26   26   + int sysinfo(int);
27   27   + int procinfo(struct pinfo *in);
25   28
26   29   // ulib.c
27   30   int stat(const char*, struct stat*);

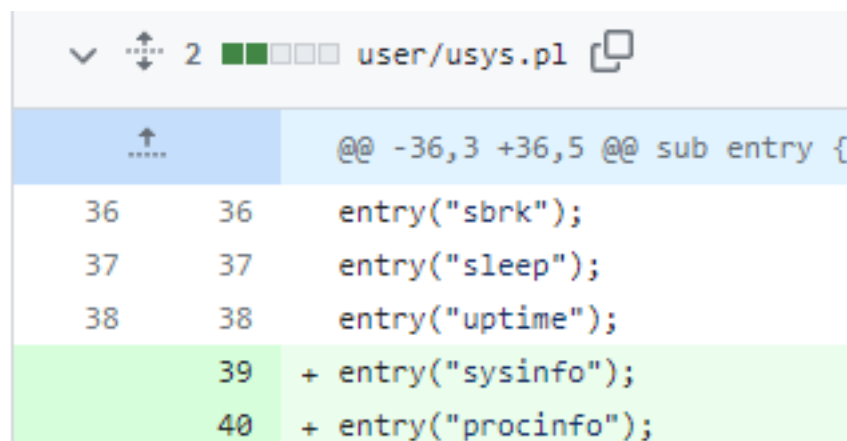
```

4.9 usys.pl

The ‘usys.pl’ script is responsible for generating system call stubs at the user level. For our ‘info’ system call:

- We updated the script to recognize and generate the necessary stub for the ‘info’ call.

This ensures that the system call is properly interfaced at the user level and can be used by user programs without any hitches.



```

↑
....  @@ -36,3 +36,5 @@ sub entry {
36    36    entry("sbrk");
37    37    entry("sleep");
38    38    entry("uptime");
39    39    + entry("sysinfo");
40    40    + entry("procinfo");

```


5 Output of the program

```
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ lab1test 65536 2
[sysinfo] active proc: 1, syscalls: 52, free pages: 32564
[procinfo 4] ppid: 3, syscalls: 10, page usage: 21
[procinfo 5] ppid: 3, syscalls: 10, page usage: 21
[sysinfo] active proc: 3, syscalls: 244, free pages: 32510
$ lab1test 65000 1
[sysinfo] active proc: 1, syscalls: 333, free pages: 32564
[procinfo 7] ppid: 6, syscalls: 10, page usage: 20
[sysinfo] active proc: 2, syscalls: 461, free pages: 32538
$
```

6 Description of XV6 Source Code Processing the ‘info’ System Call

The ‘info’ system call serves as a bridge between user programs and the kernel. Here’s a succinct walkthrough of how it’s processed within the XV6 system:

1. **User-level Invocation:** The user program invokes the ‘info’ system call using a function exposed in ‘user.h’, triggering a transition to kernel mode.
2. **Transition to Kernel Mode:** A software interrupt transitions the system from user mode to kernel mode to facilitate the privileged operations of the system call.
3. **System Call Dispatching:** Within the kernel, the system retrieves the system call number for ‘info’ and identifies the corresponding kernel function from the system call table in ‘syscall.c’.
4. **Kernel-level Execution:** The kernel executes the system call. Depending on the specifics of ‘info’, it may fetch memory statistics or process-specific data. The results are prepared for transfer back to user space.
5. **Transition to User Mode:** Post execution, the kernel readies any return values and transitions back to user mode.
6. **User-level Continuation:** The user program retrieves the results of the ‘info’ system call and proceeds with its operations.

These steps highlight the seamless integration of the ‘info’ system call within the XV6 operating system’s existing architecture.

7 Contributions of Each Member

Each member played a pivotal role in ensuring the project’s success:

- **Xiao Fan:** Comes with the initial design phase, implemented the modifications in ‘syscall.c’, and global counter for counting total system calls made and contributed to this report’s writing.

- **Kaya Gokalp:** Developed the user-level testing program, ‘sysinfotest’, modifying makefile for testing and also focused on debugging session. Also implemented proc.c with fill pinfo function and was instrumental in debugging kernel modifications.
- **Hemanth Paladugu:** Led the modifications in ‘kalloc.c’ and integrated the system call within the kernel framework with Xiao. Also, designed the method to solve num free pages functions to count the free pages available by transversing the list
- **Ashwin Nellimuttath:** Implemented the changes for the first question part 2 and 3. Implementations of number of system calls and availability of free pages in sysinfo system call. Also worked together with teammates for adding the syscall count for the process
- **NOTE:** Everyone was involved in the entire implementation process

8 Conclusion

Enhancing the XV6 operating system with the ‘info’ system call provided us with a deep dive into the intricacies of OS development. Our collective effort, coupled with thorough planning and testing, ensured the system call’s seamless integration and functionality. Through challenges and successes, this project has broadened our understanding of system call mechanics within XV6.

9 References

1. The XV6 Operating System: <https://pdos.csail.mit.edu/6.828/2012/xv6.html>
2. UNIX v6: Ritchie, D. M., and Thompson, K., "The UNIX Time-Sharing System," Communications of the ACM, July 1974.
3. adding-a-syscall-to-xv6 <https://cs631.cs.usfca.edu/guides/adding-a-syscall-to-xv6>