In God we trust

Os lab EX2

Mani Hosseini 810102552
Shayan Maleki 810102515
Hamid Mahmoudi 810102549

Introduction

1)In xv6, user programs do not trap into the kernel directly. Each system call has a small wrapper in usys.S created by the SYSCALL(name) macro. This wrapper loads the system call number (SYS_) into %eax, executes the instruction int $64 to trap into the kernel, and returns the result that the kernel places back in %eax. The functions in ulib.c provide the C-level interfaces (like write, fork, exit) that call these assembly stubs, so user code can invoke system calls just like normal C functions. This setup hides low-level details such as register use and trap handling, prevents user code from executing privileged instructions directly, and makes programs easier to write and more portable across architectures.

2) xv6 uses the int $T_SYSCALL instruction to switch from user mode to kernel mode through the Interrupt Descriptor Table (IDT). This method is easy to implement and very flexible, but it has higher overhead because the CPU performs privilege checks, pushes many registers on the stack, and looks up the handler in the IDT before jumping to the kernel.

Modern processors introduced the sysenter and sysexit instructions to reduce this overhead. These use special Model-Specific Registers (MSRs) that store the kernel entry and exit addresses, allowing a faster and more direct transition between user and kernel modes without going through the IDT.

Linux and other modern 32-bit operating systems use sysenter/sysexit when supported by the processor to speed up system calls, while xv6 still uses int for simplicity and educational purposes. On 64-bit systems, syscall and sysret are used instead.

In summary:

- int is slower but simpler, portable, and easier to study.
- sysenter/sysexit (and syscall/sysret on 64-bit) are faster because they skip some hardware checks and table lookups.
- xv6 keeps int for clarity in learning environments, while real-world systems use the faster methods to improve performance.

System call structure

3)In xv6, the system call entry in the Interrupt Descriptor Table (IDT) is defined as a Trap Gate with Descriptor Privilege Level (DPL) equal to 3, which corresponds to user-level privilege. This setup means user programs are allowed to execute the instruction int $T_SYSCALL to request kernel services.

A Trap Gate allows a controlled and synchronous transfer of execution from user mode to kernel mode without disabling hardware interrupts. This is important because system calls are intentional actions made by user programs, not unexpected events. Keeping interrupts enabled ensures the kernel can still respond to higher-priority events while servicing a system call.

Other gates in the IDT, such as those used for hardware interrupts or exceptions, have DPL set to 0. This prevents user-level code from triggering them directly, since they are reserved for the CPU or kernel to handle internal or hardware-related events. Allowing user programs to invoke those would be a major security risk, as it could let them access privileged kernel routines or corrupt system state.

This design provides a clear separation between user and kernel privileges:

- Only the system call gate (DPL=3) can be intentionally triggered by user programs.
- All other traps and interrupts (DPL=0) are restricted to kernel or hardware use.

In summary, xv6 uses a Trap Gate with DPL=3 for system calls to safely allow user programs to enter the kernel when needed, while keeping all other gates protected. This achieves both security and responsiveness in system operation.

4)When a trap such as int $64 occurs, the CPU switches from user mode (ring 3) to kernel mode (ring 0). During this privilege change, the processor automatically switches to the kernel stack and pushes the following registers onto it: SS, ESP, EFLAGS, CS, and EIP. These values record the user stack segment, user stack pointer, flags, code segment, and instruction pointer, allowing the kernel to later return to the exact point in the user program.

If the trap happens while already in kernel mode, there is no privilege change, so SS and ESP are not pushed. The CPU continues using the same stack because it is already in the correct privilege level.

After the hardware pushes these values, xv6's alltraps assembly code saves the remaining general-purpose registers (like eax, ebx, ecx, etc.) to make a complete and consistent trapframe for the current process. For software-generated traps such as system calls, the CPU does not push an error code, so alltraps adds a dummy zero to keep the trapframe format uniform.

This design ensures that the kernel can handle all types of traps—hardware interrupts, exceptions, and system calls—using a single, consistent trapframe structure.

5In xv6, functions like argint() and argptr() are used inside system call handlers to fetch the arguments that user programs passed on their stack. argint() is used for integer arguments, while argptr() is used for pointer arguments that refer to user memory.

argptr() first reads the pointer value from the user stack, then checks that the memory range it points to lies entirely within the process's valid user address space. This prevents the kernel from reading or writing outside user memory, which could otherwise cause crashes or security problems.

Without these checks, a user process could pass an invalid pointer that refers to kernel space or unmapped memory, leading to data corruption or privilege escalation. These argument-checking functions therefore make system calls both safe and reliable by validating all inputs before the kernel uses them.

6) System call arguments in xv6 follow a defined Application Binary Interface (ABI) that specifies exactly how parameters and the system call number are passed to the kernel, usually through specific registers or stack positions. The kernel expects these registers to contain valid values in a specific order so it can retrieve the arguments correctly.

If user code manually changes these registers before executing the trap instruction, the kernel will read the wrong or invalid arguments. This can cause system calls to behave incorrectly, return wrong results, crash the process, or even corrupt kernel data.

The ABI acts as a fixed contract between user programs and the kernel. Both sides must follow it for system calls to work reliably. Changing register values breaks this contract, leading to undefined or unsafe behavior. This is why operating systems enforce strict calling conventions and never allow arbitrary modification of registers before entering the kernel.

Tracing a system call with GDB

The bt command in GDB displays the **current call stack**, showing the sequence of functions that were called to reach the current point in execution.
It can help trace the **path of execution** from the current function back to the original caller.We wrote a simple program that prints the process ID using getpid(), then ran xv6 under GDB and set a breakpoint at syscall().

```
1  #include "types.h"
2  #include "user.h"
3  int main(int argc,char* argv[]) {
4    int pid = getpid();
5    printf(1, "PID: %d\n", pid);
6
7    exit();
8  }
```

```
(gdb) bt
#0  syscall () at syscall.c:135
#1  0x801069cd in trap (tf=0x8dffefb4) at trap.c:43
#2  0x8010677f in alltraps () at trapasm.S:20
#3  0x8dffefb4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb)
```

The backtrace showed that first, the kernel executes **syscall()**, which determines which system call to run based on the value in tf->eax; at this point, tf->eax holds the system call number.

```
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;
```

Next is the **trap()** function, which receives the trapframe and checks what type of trap occurred

```
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
```

Frame #2 corresponds to the **assembly-level entry point alltraps**, where the CPU jumps after executing int $64; this code saves all registers on the stack and then calls the C-level trap() function.

```
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp
```

Finally, frame #3 represents the **user process's saved trapframe**, which GDB cannot display normally because it is not a C structure but the raw register state pushed when the CPU switched from user to kernel mode.

up moves **toward earlier callers** (up the stack).
down moves **toward deeper functions** (further into the current call).
Down can not be called because we are already at Bottom frame(syscall).

```
(gdb) down
Bottom (innermost) frame selected; you cannot go down.
(gdb) up
#1  0x801069cd in trap (tf=0x8dffefb4) at trap.c:43
43          syscall();
(gdb)
```

We observed different values in myproc()->tf->eax as xv6 executed various system calls during boot and while running our program.
1 appeared first for fork(), as the system created new processes;
3 for wait(), when the parent waited for a child.
12 for sbrk(), to grow user memory.
7 for exec(), when the shell loaded our program.
11 for getpid(), our program's own system call.
multiple 16s for write(), since printf is printing text to the console.
and finally a 2 for exit(), when the program finished.
These values matched the system call numbers defined in syscall.h and showed the normal flow of process creation, execution, and termination in xv6,also PID: 3 gets printed on the console which is correct(1 is init and 2 is shell).

```
(gdb) up
#1  0x801069cd in trap (tf=0x8dffefb4) at trap.c:43
43          syscall();
(gdb) print tf->eax
$1 = 1
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135       struct proc *curproc = myproc();
(gdb) print tf->eax
No symbol "tf" in current context.
(gdb) print tf->eax
No symbol "tf" in current context.
(gdb) print myproc()->tf->eax
$2 = 3
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135       struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$3 = 12
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135       struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$4 = 7
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135       struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$5 = 11
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135       struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$6 = 16
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135       struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$7 = 16
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135       struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$8 = 16
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135       struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$9 = 16
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135       struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$10 = 16
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135       struct proc *curproc = myproc();
```

```
Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135       struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$10 = 16
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135       struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$11 = 16
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135       struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$12 = 16
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135       struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$13 = 2
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135       struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$14 = 16
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135       struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$15 = 16
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135       struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$16 = 5
```

Simple arithmetic system call

First we define the system call in syscall.h

```
#define SYS_simple_arithmetic 22    // from EX2
```

Then we define the function and add it to syscall table

```
extern int sys_simple_arithmetic(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_simple_arithmetic] = sys_simple_arithmetic,

};
```

After that we define it in user.h and add an stub for it in assembly

```
You, 2 minutes ago | 2 authors (hmdmhmd2004 and one other)
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 simple_arithmetic_syscall(int a, int b);

```

In the usys.S file, we first include the headers syscall.h and traps.h so the assembler knows the system call numbers and interrupt constants defined in xv6. Then,we save the EBX register on the stack to preserve its original value. Next, we move the first and second function arguments from the stack into registers. We then load the system call number into EAX and execute the software interrupt to trap into the kernel. Finally, we restore EBX and return to the user program.

```
#include "syscall.h"
#include "traps.h"

.globl simple_arithmetic_syscall
simple_arithmetic_syscall:
    pushl %ebx              # preserve callee-saved register
    movl 8(%esp), %ebx      # a  (after pushl, first arg is at 8(%esp))
    movl 12(%esp), %ecx     # b
    movl $SYS_simple_arithmetic, %eax
    int $T_SYSCALL                You, 9 minutes ago • Uncommitted changes
    popl %ebx
    ret
```

And finally we make a user side program and run it.

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

int
main(int argc, char *argv[])
{
  int a = 5, b = 3;
  if (argc == 3) {
    a = atoi(argv[1]);
    b = atoi(argv[2]);
  }
  int r = simple_arithmetic_syscall(a, b);
  printf(1, "user: (%d+%d)*(%d-%d) = %d\n", a, b, a, b, r);
  exit();
}
```

```
$ arithmatic
Calc: (5+3)*(5-3) = 16
user: (5+3)*(5-3) -> 16
```

1.Make duplicate System call

First we do the usual routine of adding a new system call like previous section:

User.h:

```c
int make_duplicate(const char *src_file);
```

Syscall.h:

```c
#define SYS_make_duplicate 23
```

Syscall.c:

```
extern int sys_make_duplicate(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_simple_arithmetic] = sys_simple_arithmetic,

    [SYS_make_duplicate] sys_make_duplicate,
```

usys.S:

```
21  SYSCALL(mknod)
22  SYSCALL(unlink)
23  SYSCALL(fstat)
24  SYSCALL(link)
25  SYSCALL(mkdir)
26  SYSCALL(chdir)
27  SYSCALL(dup)
28  SYSCALL(getpid)
29  SYSCALL(sbrk)
30  SYSCALL(sleep)
31  SYSCALL(uptime)
32  SYSCALL(make_duplicate)
33
34
```

And finally we add the function itself to sysfile.c and make a user program to test it.

```c
int sys_make_duplicate(void)
{
  char *src_path;
  if (argstr(0, &src_path) < 0)
    return 1;

  char dst_path[MAXPATH];
  int n = strlen(src_path);
  const char *suf = "_copy";
  if (n + strlen(suf) >= MAXPATH)
    return 1;

  // make the copy's name
  safestrcpy(dst_path, src_path, MAXPATH);
  safestrcpy(dst_path + n, suf, MAXPATH - n);

  begin_op(); // filesystem stuff start and end with these.

  // Look up source
  struct inode *src = namei(src_path);
  if (src == 0)
  {
    end_op();
    return -1;
  }

  ilock(src);
  if (src->type != T_FILE)
  { // check file type
    iunlockput(src);
    end_op();
    return 1;
  }

  // Destination must not already exist
  struct inode *tmp = namei(dst_path);
  if (tmp)
  {
    iunlockput(tmp);
    iunlockput(src);
    end_op();
    return 1;
  }

  // (create() returns a *locked* inode on success)
  struct inode *dst = create(dst_path, T_FILE, 0, 0);
  if (dst == 0)
  {
    iunlockput(src);
    end_op();
    return 1;
  }

  // Copy loop
  int rc = 0;
  char *buf = kalloc(); // kalloc() returns a kernel page buffer (size PGSIZE).
  if (buf == 0)
  {
    iunlockput(dst);
    iunlockput(src);
    end_op();
    return 1;
  }

  uint offset = 0;
  while (1)
  {
    int nread = readi(src, buf, offset, PGSIZE);
    if (nread < 0)
    {
      rc = 1;
      break;
    }
    if (nread == 0)
      break;

    int nwrite = writei(dst, buf, offset, nread);
    if (nwrite != nread)
    {
      rc = 1;
      break;
    }
    offset += nread;
  }      You, now • Uncommitted changes
```

```c
#include "types.h"      You, 2 minutes ago • Uncommitte
#include "user.h"

int main(int argc, char *argv[]) {
  if(argc < 2){
    printf(1, "usage: make_duplicate filename\n");
    exit();
  }

  int r = make_duplicate(argv[1]);
  exit();
}
```

```
Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ni
t 58
init: starting sh
Hamid, Mani, SHAYAN
$ echo hello > test
$ make_duplicate test
$ cat test_copy
hello
$ _
```

In the next three parts some states were common, it includes:

Adding the definition in syscall.h, adding a pointer in syscall.c, the main code for each system call where in two files, proc.c and sysfile.c for the grep will be check in the part its from, changing usys.s and user.h for the user level after changing the previous parts and a test.c file for each of the system calls which also will be mentioned in the correct part and the makefile which adds two lines per system call. So for the common parts we have:

Adding these lines at the end of syscall.h:

```
25  #define SYS_show_process_family 24
26  #define SYS_grep_syscall 25
27  #define SYS_set_priority_syscall 26
```

Adding 2 lines per each system call in syscall.c:

```
104  extern int sys_show_process_family(void);
105  extern int sys_grep_syscall(void);
106  extern int sys_set_priority_syscall(void);
```

```
132      [SYS_show_process_family] sys_show_process_family,
133      [SYS_grep_syscall] sys_grep_syscall,
134      [SYS_set_priority_syscall]  sys_set_priority_syscall,
```

Adding 3 lines in usys.S:

```
33  SYSCALL(show_process_family)
34  SYSCALL(grep_syscall)
35  SYSCALL(set_priority_syscall)
```

Adding 3 lines in user.h for the calls from user level and testing:

```
28  int show_process_family(int);
29  int grep_syscall(const char*, const char*, char*, int);
30  int set_priority_syscall(int pid, int priority);
```

Adding 2 lines per system call in makefile so it makes the files for test and also cleaning them in make clean:

```
168  UPROGS=\
169      _cat\
170      _echo\
171      _forktest\
172      _grep\
173      _init\
174      _kill\
175      _ln\
176      _ls\
177      _mkdir\
178      _rm\
179      _sh\
180      _stressfs\
181      _usertests\
182      _wc\
183      _zombie\
184      _find_sum\
185      _pid\
186      _arithmatic\
187      _make_duplicate\
188      _showfamilytest\
189      _grep_test\
190      _priority_test\
```

```
257  EXTRA=\
258      mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
259      ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
260      printf.c umalloc.c\
261      README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
262      .gdbinit.tmpl gdbutil\
263      showfamilytest.c\
264      grep_test.c\
265      priority_test.c\
```

Now we can check the functionality of the codes and parts they change for each system call:

2.Show process family

This system call inspects the process table to identify and display the parent, children, and siblings of the process specified by the “pid” argument.

In all three parts we first need to fetch the input:

```
602  int
603  sys_show_process_family(void)
604  {
605    struct proc *p;
606    struct proc *target_proc = 0;
607    struct proc *parent_proc = 0;
608    int pid;
609    int has_children = 0;
610    int has_siblings = 0;
611
612    if(argint(0, &pid) < 0)
613      return -1;
614
615    acquire(&ptable.lock);
```

If we don’t have it we get an error, we then start our job after getting the pid.

With pid we first have to find the actual process, it is simply done for a for on all processes and checking its pid, if its not founder get error and release(it is important since we have used aquire):

```
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->pid == pid){
    target_proc = p;
    break;
  }
}

if(target_proc == 0){
  release(&ptable.lock);
  cprintf("Process with pid %d not found.\n", pid);
  return -1;
}
```

Else we simply check for parents of each process and if its pid we can print that as a child, and before that check for the parent of our current process and print it, and then use another for on all processes and if they have the same parent as our current process they are siblings, and

like that this part is done and we then release:

```
638     cprintf("Children of process %d:\n", pid);
639     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
640       if(p->parent == target_proc){
641         cprintf("Child pid: %d\n", p->pid);
642         has_children = 1;
643       }
644     }
645     if(!has_children){
646       cprintf("(No children)\n");
647     }
648
649     cprintf("Siblings of process %d:\n", pid);
650     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
651       if(parent_proc && p->parent == parent_proc && p->pid != pid){
652           cprintf("Sibling pid: %d\n", p->pid);
653           has_siblings = 1;
654       }
655     }
656     if(!has_siblings){
657       cprintf("(No siblings)\n");
658     }
659
660     release(&ptable.lock);
661     return 0;
662   }
```

Here's the test for it which is in showfamilytest.c

```
1   #include "types.h"
2   #include "stat.h"
3   #include "user.h"
4
5   int
6   main(int argc, char *argv[])
7   {
8       int parent_pid = getpid();
9       int child1_pid = -1;
10
11      if(fork() == 0) {
12          child1_pid = getpid();
13
14          if (fork() == 0) {
15              sleep(60);
16              exit();
17          }
18
19          sleep(60);
20          exit();
21      }
22
23      if(fork() == 0) {
24
25          sleep(10);
26
27
28          show_process_family(parent_pid);
29
30          sleep(60);
31          exit();
32      }
33
34      wait();
35      wait();
36      wait();
37      exit();
38  }
39
```

```
$ showfamilytest
My id: 3, My parent id: 2
Children of process 3:
Child pid: 4
Child pid: 5
Siblings of process 3:
(No siblings)
```

3.Grep

For this part we are tasked to find a key word in a file and if found print the whole line in which it was found.

To do this we add the system call function to sysfile for ease of access and then read the arguments such as the key word, file name and buffer size and the users buffer:

```
int
sys_grep_syscall(void)
{
  char *keyword, *filename, *user_buffer;
  int buffer_size;
  struct inode *ip;
  char *kernel_buf = 0;
  int result = -1;

  if (argstr(0, &keyword) < 0 || argstr(1, &filename) < 0 || argint(3, &buffer_size) < 0)
    return -1;
  if (argptr(2, &user_buffer, buffer_size) < 0)
    return -1;
```

After that we begin the operation and start by reading the file which was asked.

```
  begin_op();

  if ((ip = namei(filename)) == 0) {
    end_op();
    return -1;
  }
  ilock(ip);

  if (ip->size == 0 || (kernel_buf = kalloc()) == 0) {
    iunlockput(ip);
    end_op();
    return -1;
  }

  int bytes_to_read = (ip->size < PGSIZE) ? ip->size : (PGSIZE - 1);
  if (readi(ip, kernel_buf, 0, bytes_to_read) != bytes_to_read) {
    kfree(kernel_buf);
    iunlockput(ip);
    end_op();
    return -1;
  }
  kernel_buf[bytes_to_read] = '\0';

  iunlockput(ip);
  end_op();
```

For each line we need to check for the keywords, so after reading a line in the while we then use a function to check if the keyword exists in the line:

```
617   while(current_pos < kernel_buf + bytes_to_read) {
618     char *line_end = current_pos;
619     while(line_end < kernel_buf + bytes_to_read && *line_end != '\n') {
620       line_end++;
621     }
622
623     char temp_char = *line_end;
624     *line_end = '\0';
625
626     if (strstr(line_start, keyword)) {
627       int line_len = strlen(line_start);
628       if(line_len > buffer_size - 1)
629           line_len = buffer_size - 1;
630
631       *line_end = temp_char;
632
633       if(copyout(myproc()->pgdir, (uint)user_buffer, line_start, line_len) == 0){
634         if(*line_end == '\n' && (line_len + 1 < buffer_size)){
635           copyout(myproc()->pgdir, (uint)user_buffer + line_len, "\n", 1);
636           copyout(myproc()->pgdir, (uint)user_buffer + line_len + 1, "\0", 1);
637         } else {
638           copyout(myproc()->pgdir, (uint)user_buffer + line_len, "\0", 1);
639         }
640         result = 0;
641       }
642       break;
643     }
644
645     *line_end = temp_char;
646     line_start = line_end + 1;
647     current_pos = line_start;
648   }
649
650   kfree(kernel_buf);
651   return result;
652 }
```

```
552
553   static char*
554   strstr(const char *haystack, const char *needle)
555   {
556     int i, j;
557     int needle_len = strlen(needle);
558     int haystack_len = strlen(haystack);
559
560     if (needle_len == 0)
561       return (char*)haystack;
562
563     for (i = 0; i <= haystack_len - needle_len; i++) {
564       for (j = 0; j < needle_len; j++) {
565         if (haystack[i+j] != needle[j])
566           break;
567       }
568       if (j == needle_len)
569         return (char*)haystack + i;
570     }
571     return 0;
572   }
```

And after finishing with either found which turns result into 0 or failure which is the normal value for result with -1, we return and if the result was 0 we can then read the line which was written into users buffer in the same while which found the keyword in the line, here's a test and it's results:

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

int
main(int argc, char *argv[])
{
  char buf[256];

  if(argc != 3){
    printf(2, "Usage: manual_grep keyword filename\n");
    exit();
  }

  char* keyword = argv[1];
  char* filename = argv[2];

  printf(1, "Searching for '%s' in file '%s'...\n", keyword, filename);

  int ret = grep_syscall(keyword, filename, buf, sizeof(buf));

  if (ret == 0) {
    printf(1, "Success! Found line: %s", buf);
  } else {
    printf(2, "Failure. Syscall returned %d.\n", ret);
  }

  exit();
}
```

```
$ echo A line with keyword YIPPEE > file1
$ echo A line without that keyword > file2
$ grep_test YIPPEE file1
Searching for 'YIPPEE' in file 'file1'...
Success! Found line: A line with keyword YIPPEE
$ grep_test YIPPEE file2
Searching for 'YIPPEE' in file 'file2'...
Failure. Syscall returned -1.
$
```

4.Set priority

For this one we first add a new arg to our process called priority which is an enum with default of normal, and can be changed into high or low as well using the system call.

```
37  struct proc {
38    uint sz;                     // Size of process memory (bytes)
39    pde_t* pgdir;                // Page table
40    char *kstack;                // Bottom of kernel stack for this process
41    enum procstate state;        // Process state
42    int pid;                     // Process ID
43    int priority;                // process priority
44    struct proc *parent;         // Parent process
45    struct trapframe *tf;        // Trap frame for current syscall
46    struct context *context;     // swtch() here to run process
47    void *chan;                  // If non-zero, sleeping on chan
48    int killed;                  // If non-zero, have been killed
49    struct file *ofile[NOFILE];  // Open files
50    struct inode *cwd;           // Current directory
51    char name[16];               // Process name (debugging)
52  };
```

We will be adding/changing two main parts for it to work, one is just adding the system call itself in proc.c which simply searches up a pid and if it finds the process with that pid is changes its priority and if not return -1 as error:

```
665  int
666  sys_set_priority_syscall(void)
667  {
668    int pid, priority;
669    struct proc *p;
670    int found = 0;
671
672    if(argint(0, &pid) < 0 || argint(1, &priority) < 0)
673      return -1;
674
675    if(priority < PRI_HIGH || priority > PRI_LOW)
676      return -1;
677
678    acquire(&ptable.lock);
679    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
680      if(p->pid == pid){
681        p->priority = priority;
682        found = 1;
683        break;
684      }
685    }
686    release(&ptable.lock);
687
688    if(found)
689    return 0;
690      else
691    return -1;
692  }
```

The main part we will change is the scheduling, it's the part that tells which process should be ran, we just need to modify it so it starts with high priority ones first, for this we add a done signal so when it finds and switched in a for it wont for the rest of the for's and first tries to finish the process with the highest priority, it makes it so it first tries to finish high_pro then normal_pro and finally low_pro

For high pro and start of it we have:

```
399     if (done1 == 0) {
400       for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
401         if(p->state != RUNNABLE || p->priority != PRI_NORMAL)
402           continue;
403         done2 = 1;
404         c->proc = p;
405         switchuvm(p);
406         p->state = RUNNING;
407         swtch(&(c->scheduler), p->context);
408         switchkvm();
409         c->proc = 0;
410       }
411     }
412
413     if (done1 == 0 && done2 == 0) {
414       for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
415         if(p->state != RUNNABLE || p->priority != PRI_LOW)
416           continue;
417         c->proc = p;
418         switchuvm(p);
419         p->state = RUNNING;
420         swtch(&(c->scheduler), p->context);
421         switchkvm();
422         c->proc = 0;
423       }
424     }
```

Then for normal and low we have:

And finally we release:

```
425       release(&ptable.lock);
426     }
427   }
```

So with that we are done setting priorities.

For testing since a video can show it better but its not simple to add a video we just explain what was seen by running the test and it can be tested in person later for better understanding of the result:

We first introduce a time_consuming task and define the priorities:

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

#define PRI_HIGH   0
#define PRI_NORMAL 1
#define PRI_LOW    2

void cpu_intensive_task(int priority_level, int pid) {
char* priority_str = "";
if (priority_level == PRI_HIGH) priority_str = "High";
if (priority_level == PRI_LOW) priority_str = "Low";

volatile unsigned long long i;
for (i = 0; i < 500000000; i++);

printf(1, "--> Child PID %d (Set to %s Priority) finished.\n", pid, priority_str);
}
```

The we use fork to make two children with that task:

```c
int main(void) {
int pid1, pid2;


pid1 = fork();
if (pid1 == 0) {
    cpu_intensive_task(PRI_HIGH, getpid());
    exit();
}

pid2 = fork();
if (pid2 == 0) {
    cpu_intensive_task(PRI_LOW, getpid());
    exit();
}
```

And in the end we set the priority for one to low and one to high and wait for them to finish:

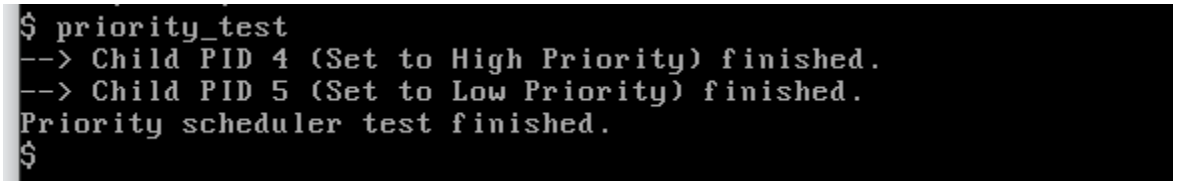
```c
set_priority_syscall(pid2, PRI_LOW);
set_priority_syscall(pid1, PRI_HIGH);


wait();
wait();

printf(1, "Priority scheduler test finished.\n");
exit();
}
```

With that the one with higher priority should end much faster than the other one, in fact the other one should not progress while the high priority is running:

(These two images were taken with around few second in between which means the high priority finished much slower since just by setting the priority of the first task to high)



And thus this project is finished :)