Lab: system calls

https://pdos.csail.mit.edu/6.828/2021/labs/syscall.html

How syscall is compiled

1. System Call Interface

The system call interface in xv6 is implemented through a combination of user-space stub functions, a system call dispatch mechanism, and the kernel's implementation of the system calls.

2. User Stub Functions

User stub functions are provided in user space to make system calls. These stubs are typically defined in `user.h` and `usys.S` files.

`user.h`

The `user.h` file declares the prototypes for the system call functions:

```
cCopy code// user.h

int fork(void);
int exit(int status) __attribute__((noreturn));
int wait(int *status);
int pipe(int fds[2]);
int write(int fd, const void *buf, int n);
int read(int fd, void *buf, int n);
int close(int fd);
int kill(int pid);
int exec(const char *path, char *const argv[]);
int open(const char *path, int flags);
int mknod(const char *path, short major, short minor);
int unlink(const char *path);
int fstat(int fd, struct stat *st);
int link(const char *old, const char *new);
int mkdir(const char *path);
int chdir(const char *path);
int dup(int fd);
int getpid(void);
char* sbrk(int n);
int sleep(int n);
int uptime(void);
```

`usys.S`

The `usys.S` file contains the assembly code for the user stub functions. These functions set up the necessary registers and invoke the `syscall` instruction to switch to kernel mode and perform the system call.

```
assemblyCopy code// usys.S
```

```
.globl sys_fork
sys_fork:
    li a7, SYS_fork
    ecall
    ret

.globl sys_exit
sys_exit:
    li a7, SYS_exit
    ecall
    ret

// Repeat for all other syscalls...
```

3. System Call Numbers

System call numbers are defined in `syscall.h` and correspond to each system call. These numbers are used to identify the system call in the `ecall` instruction.

```
cCopy code// syscall.h

#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
```

4. Compilation Process

During the compilation process, the `user.h` and `usys.S` files are compiled along with the rest of the user programs. The Makefile in xv6 ensures that these files are included in the build process.

Makefile

The Makefile includes rules to compile the user stub functions and link them with the user programs.

```
makefileCopy code# Makefile

UPROGS = _cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs
_usertests _wc _zombie

$(UPROGS): %: %.o ulib.o user.o printf.o umalloc.o
    ld -m elf_i386 -N -e main -Ttext 0 -o $@ $^
    nm $@ | grep -v U | sort > $*.sym

%.o: %.c
    gcc -m32 -fno-pic -static -ggdb -fno-common -fno-builtin -MD -Wall -Werror -O
-I. -c -o $@ $<
```

5. User Program Compilation

User programs are compiled and linked with the user stub functions and the necessary libraries (`ulib.o`, `user.o`, `printf.o`, `umalloc.o`). This ensures that the system calls can be invoked from user space.

System call tracing

Modify the struct of process in `kernal/proc.h`:

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

  // p->lock must be held when using these:
  enum procstate state; // Process state
  void *chan;           // If non-zero, sleeping on chan
  int killed;           // If non-zero, have been killed
  int xstate;           // Exit status to be returned to parent's wait
  int pid;              // Process ID

  // wait_lock must be held when using this:
  struct proc *parent; // Parent process

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int tracemask;
};
```

By adding a new field called `tracemask`, kernel can determine if it is required to print trace info when completing syscalls.

Also, we need to add another syscall called `trace` that can set the `tracemask` of process.

In kernel/syscall.c:

```
void syscall(void)
{
  int num;
  struct proc *p = myproc();

  num = p->trapframe->a7;
  if (num > 0 && num < NELEM(syscalls) && syscalls[num])
  {
    p->trapframe->a0 = syscalls[num](); // a0 is the return value
    if ((p->tracemask >> num) % 2 == 1)
    {
      printf("syscall %s -> %d\n", syscallnames[num], p->trapframe->a0);
    }
  }
  else
  {
    printf("%d %s: unknown sys call %d\n",
           p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}

uint64 sys_trace(void)
{
  int mask;
  argint(0, &mask);
  struct proc *p = myproc();
  p->tracemask = mask;
  return 0;
}
```

Result

```
$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3
-nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-
device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ trace 32 grep hello README
syscall read -> 1023
syscall read -> 968
```

```
syscall read -> 235
syscall read -> 0
$ trace 2147483647 grep hello README
syscall trace -> 0
syscall exec -> 3
syscall open -> 3
syscall read -> 1023
syscall read -> 968
syscall read -> 235
syscall read -> 0
syscall close -> 0
$ grep hello README
$ trace 2 usertests forkforkfork
usertests starting
syscall fork -> 7
test forkforkfork: syscall fork -> 8
OK
syscall fork -> 69
ALL TESTS PASSED
```

System call sysinfo

In `kernel/kalloc.c`, add another function that counts free memory in bytes:

```
int freeMemInBytes(void)
{
  struct run *cur = kmem.freelist;
  int count = 0;
  while (cur)
  {
    count++;
    cur = cur->next;
  }
  return count * 4096;
}
```

in kernel/proc.c, add another function that counts number of processes with state `UNUSED`

```
int numOfProc(void)
{
  struct proc *p;
  int count = 0;
  for (p = proc; p < &proc[NPROC]; p++)
  {
    acquire(&p->lock);
    if (p->state != UNUSED)
    {
      count++;
    }
    release(&p->lock);
  }
  return count;
```

```
}
```

Perform the operations in above section to add stub code and syscall number so that the program can compile.

In kernel/syscall.c, add the function that implements `sysinfo`:

```
uint64 sys_sysinfo(void)
{
  uint64 addr;
  argaddr(0, &addr);
  struct sysinfo info;

  info.freemem = freeMemInBytes();
  info.nproc = numOfProc();
  struct proc *p = myproc();
  if (copyout(p->pagetable, addr, (char *)&info, sizeof(info)) < 0)
  {
    return -1;
  }
  return 0;
}
```

Result

I added some line to print debugging info and can be omitted.

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ sysinfotest
sysinfotest: start
sysinfotest: OK
$
```