SWE3004 Operating Systems, Spring 2025

Project 1. System Call

TA)

Gwanjong Park

Yunseong Shin

Project Plan

- Total 6 projects
  - ~~0) Booting xv6 operating system~~
  - 1) System call
  - 2) CPU scheduling
  - 3) Virtual memory
  - 4) Page replacement
  - 5) File systems

Project 1: Make system calls

- Goal: make five new system calls (getnice, setnice, ps, meminfo, waitpid)

- Synopsis 개요
  - int getnice(int pid);
  - int setnice(int pid, int value);
  - void ps(int pid);
  - uint64 meminfo(void);
  - int waitpid(int pid);

Project 1: Make system calls

- Description
  - The ***getnice*** function obtains the nice value of a process.
  - The ***setnice*** function sets the nice value of a process.
  - The **default nice value is 20**. Lower nice values cause more favorable scheduling.
  - It will be necessary to implement the nice value before creating the system call.
  - **The range of valid nice value is 0~39**

Project 1: Make system calls

- Description (cont'd)
  - In kernel, the ***ps*** system call prints out process(s)'s information, which includes name, pid, state and priority(nice value) of each process.
  - If the pid is 0, print out all processes' information.
  - Otherwise, print out corresponding process's information.
  - If there is no process corresponding to the pid, print out nothing.

```
name        pid         state               priority
init        1           SLEEPING            20
sh          2           SLEEPING            20
ps          3           RUNNING             20
```

  - ***meminfo*** prints available memory in bytes.
  - ***waitpid*** suspends execution until the specified process terminates.

Project 1: Make system calls

- Return values
  - **getnice** : Return the nice value of target process on success. Return -1 if there is no process corresponding to the pid.
  - **setnice** : Return 0 on success. Return -1 if there is no process corresponding to the pid or the nice value is invalid.
  - **ps :** No return value. Prints the process list directly to the console.
  - **meminfo**: Returns the amount of free memory (in bytes) available in the system.
  - **waitpid**: Returns 0 when the specified process terminates successfully. Returns -1 if the process does not exist or if the calling process does not have permission to wait for it.

How to define system call

1. Add your syscall to user/usys.pl

```
entry("fork");
entry("exit");
entry("wait");
entry("pipe");
```

2. Add syscall number to kernel/syscall.h

```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
```

3. Add extern and syscall element in kernel/syscall.c

```
// Prototypes for the functions that handle system calls.
extern uint64 sys_fork(void);
extern uint64 sys_exit(void);
extern uint64 sys_wait(void);
extern uint64 sys_pipe(void);
```

```
// An array mapping syscall numbers from syscall.h
// to the function that handles the system call.
static uint64 (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
```

How to define system call

4. Add a sys_ function to kernel/sysproc.c

```
uint64
sys_fork(void)
{
  return fork();
}
```

5. Add a function that performs an action to kernel/proc.c

```
// Create a new process, copying the parent.
// Sets up child kernel stack to return as if from fork() system call.
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *p = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy user memory from parent to child.
```

How to define system call

6. Add a definition to kernel/defs.h and user/user.h

```
// proc.c
int             cpuid(void);
void            exit(int);
int             fork(void);
```

```
// system calls
int fork(void);
```

How to test your system call

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

int main()
{
        int i;
        for (i=1; i<11; i++) {
                printf(1, "%d: ", i);
                if (getpname(i))
                        printf(1, "Wrong pid\n");
        }

        exit();
}
```

mytest.c (just example)

“user/mytest.c” is an example code.
Create and use your own test code.

```makefile
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/_mytest\
```

Makefile

Test with user program

```
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -global virtio-mmio.for
ce-legacy=false -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 1 starting
hart 2 starting
init: starting sh
Student ID: 2024123456
Name: Gildong Hong
===========Your message==========
$ mytest
>>>Testing getpname:
init
>>>Testing getnice and setnice:
initial nice value: 20
nice value after setting: 10
>>>Testing ps:
name            pid     state           priority
init            1       SLEEPING        20
sh              2       SLEEPING        20
mytest          3       RUNNING         10
>>>Testing meminfo:
available memory: 133263360 bytes
>>> Testing waitpid:
wait
start1
start2
end1
done1 4 10
end2
done2 5 10
$
```

Submission

- This project is to implement only the system calls (getnice, setnice, ps, meminfo, waitpid)
  - The user program for testing is irrelevant.
- Use the ***submit*** & ***check-submission*** binary file in Ye Server
  - $ make clean
  - $ ~swe3004/bin/submit pa1 xv6-riscv
  - You can submit several times, and the submission history can be checked through check-submission
    - Only the last submission will be graded

Submission

- PLEASE DO NOT COPY
  - We will run inspection program on all the submissions
  - Any unannounced penalty can be given to **both students**
    - 0 points / negative points / F grade …

- Due date: 4/1(Tue.), 23:59:59 PM
  - **-25%** per day for delayed submission

Questions

- If you have questions, please ask on i-campus
  - Please use the discussion board
  - Discussion board preferred over messages

- You can also visit Corporate Collaboration Center #85533
  - Please iCampus message TA before visiting

- Reading xv6 commentary will help you a lot

https://pdos.csail.mit.edu/6.828/2023/xv6/book-riscv-rev3.pdf

Appendix. Trap Handling Process on xv6

- How RISC-V Handles Traps and System Calls
- System calls are executed using **ecall**.
- The CPU jumps to the kernel's trap handler, located in **stvec**.

Appendix. Trap Handling Process on xv6

- Example : kill system call

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

int
main(int argc, char **argv)
{
  int i;

  if(argc < 2){
    fprintf(2, "usage: kill pid...\n");
    exit(1);
  }
  for(i=1; i<argc; i++)
    kill(atoi(argv[i]));
  exit(0);
}
```

kill.c (user level)

```c
// system calls
int fork(void);
int exit(int) __attribute__((noreturn));
int wait(int*);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
```

user.h

Functions defined as assembly

```
print "#include \"kernel/syscall.h\"\n";

sub entry {
    my $name = shift;
    print ".global $name\n";
    print "${name}:\n";
    print " li a7, SYS_${name}\n";
    print " ecall\n";
    print " ret\n";
}
```

```
entry("kill");
```

usys.pl

```
$U/usys.S : $U/usys.pl
	perl $U/usys.pl > $U/usys.S

$U/usys.o : $U/usys.S
	$(CC) $(CFLAGS) -c -o $U/usys.o $U/usys.S
```

Makefile

```
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
```

syscall.h

subroutine entry → li a7, SYS_kill → ecall → ??? → ret

ecall → uservec → usertrap

```
.section trampsec
.globl trampoline
.globl usertrap
trampoline:
.align 4
.globl uservec
uservec:
        #
        # trap.c sets stvec to point here, so
        # traps from user space start here,
        # in supervisor mode, but with a
        # user page table.
        #

        # save user a0 in sscratch so
        # a0 can be used to get at TRAPFRAME.
        csrw sscratch, a0

        # each process has a separate p->trapframe memory area,
        # but it's mapped to the same virtual address
        # (TRAPFRAME) in every process's user page table.
        li a0, TRAPFRAME
        # save the user a0 in p->trapframe->a0
        csrr t0, sscratch
        sd t0, 112(a0)

        # initialize kernel stack pointer, from p->trapframe->kernel_sp
        ld sp, 8(a0)

        # make tp hold the current hartid, from p->trapframe->kernel_hartid
        ld tp, 32(a0)

        # load the address of usertrap(), from p->trapframe->kernel_trap
        ld t0, 16(a0)

        # fetch the kernel page table address, from p->trapframe->kernel_satp.
        ld t1, 0(a0)

        # wait for any previous memory operations to complete, so that
        # they use the user page table.
        sfence.vma zero, zero

        # install the kernel page table.
        csrw satp, t1

        # flush now-stale user entries from the TLB.
        sfence.vma zero, zero

        # jump to usertrap(), which does not return
        jr t0
```

trampoline.S

Trap cause → scause register

```c
//
// handle an interrupt, exception, or system call from user space.
// called from trampoline.S
//
void
usertrap(void)
{
  int which_dev = 0;

  if((r_sstatus() & SSTATUS_SPP) != 0)
    panic("usertrap: not from user mode");

  // send interrupts and exceptions to kerneltrap(),
  // since we're now in the kernel.
  w_stvec((uint64)kernelvec);
```

```c
if(r_scause() == 8){
  // system call

  if(killed(p))
    exit(-1);

  // sepc points to the ecall instruction,
  // but we want to return to the next instruction.
  p->trapframe->epc += 4;

  // an interrupt will change sepc, scause, and sstatus,
  // so enable only now that we're done with those registers.
  intr_on();

  syscall();
```

trap.c

syscall

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

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

```c
// An array mapping syscall numbers from syscall.h
// to the function that handles the system call.
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,
```

syscall.c

sys_kill → kill

```c
uint64
sys_kill(void)
{
  int pid;

  argint(0, &pid);
  return kill(pid);
}
```

sysproc.c

```c
// Kill the process with the given pid.
// The victim won't exit until it tries to return
// to user space (see usertrap() in trap.c).
int
kill(int pid)
{
  struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++){
    acquire(&p->lock);
    if(p->pid == pid){
      p->killed = 1;
      if(p->state == SLEEPING){
        // Wake process from sleep().
        p->state = RUNNABLE;
      }
      release(&p->lock);
      return 0;
    }
    release(&p->lock);
  }
  return -1;
}
```

proc.c

usertrap → usertrapret

```c
  syscall();
} else if((which_dev = devintr()) != 0){
  // ok
} else {
  printf("usertrap(): unexpected scause 0x%lx pid=%d\n", r_scause(), p->pid);
  printf("            sepc=0x%lx stval=0x%lx\n", r_sepc(), r_stval());
  setkilled(p);
}

if(killed(p))
  exit(-1);

// give up the CPU if this is a timer interrupt.
if(which_dev == 2)
  yield();

usertrapret();
```

trap.c, usertrap()

```c
//
// return to user space
//
void
usertrapret(void)
{
  struct proc *p = myproc();

  // we're about to switch the destination of traps from
  // kerneltrap() to usertrap(), so turn off interrupts until
  // we're back in user space, where usertrap() is correct.
  intr_off();

  // send syscalls, interrupts, and exceptions to uservec in trampoline.S
  uint64 trampoline_uservec = TRAMPOLINE + (uservec - trampoline);
  w_stvec(trampoline_uservec);
```

trap.c, usertrapret()