OS_assignment_12754_2019092824

Before I make a system call...

> *How can I make a new System call? How can I implement them? To make a system call that returns ppid, I have to know What is going on inside my machine.*

I started from `user.h`.

`/user/user.h`

```
// Function prototype for system calls, They are collected in usys.S
int fork(void);

int exit(int) __attribute__((noreturn));
// compiler magic to tell the compiler that the function will never return

int wait(int*);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(const 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);
```

- These System calls are collected in `usys.S` , which is written in assembly code
- `usys.S` is made automatically by `usys.pl`, which is a generator for assembly code using macros. So Lets see `usys.pl`.

`/user/usys.pl`

```
#!/usr/bin/perl -w

# Generate usys.S, the stubs for syscalls.
# Generate an assembly code that includes system calls.

print "# generated by usys.pl - do not edit\n";

print "#include \"kernel/syscall.h\"\n";
# It includes syscall.h which is just a series of definition, which is basically just an association a system calls to a number.
```

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

entry("fork");
...
```

- This pl file automatically generates an assembly code with macro.
    - The prototype of assembly code is written in `entry`, So basically, all of system calls shares same production rules.
- The result can be seen on `usys.S`, and here is a basic prototype of `usys.S`

```
# generated by...
#include "kernel/syscall.h"

...

.global open
open :
    li a7, SYS_open
    # Thanks to syscall.h, it is just a uint.
    # So basically it is an instruction that stores System call number
    to a register a7.

    ecall
    # ecall instruction switches usermode to kernel mode, and kernel
    will execute a code.
    # The kernel's trap handler inspects the system call number and
    arguments(a0~) , performs the requested operation, and places the
    result in a designated register. In this case, a0.

    ret
    # ret instruction switches kernel mode to usermode, and returns
    a result of kernel mode's execution.
    # Once the kernel finishes handling the system call, it uses
    the ret instruction
    to return to user mode and resume execution of the program.
...
```

- Let's have a closer look in `ecall` and `ret` instruction in detail.

`ecall` instruction in RISC-V system

> *ecall : environment call*

1. It traps the processor
2. It can be used to switch between privilege mode
3. It is used to implement system call to pass/access system resources.

After `ecall` ... `trampoline.S`

According to `lab03.pdf`, `ecall` instruction changes the priviledge level (user to supervisor), and then It jumps to `uservec` in `trampoline.S`

```
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 registers in TRAPFRAME
        sd ra, 40(a0)
        sd sp, 48(a0)
        sd gp, 56(a0)
        sd tp, 64(a0)
        sd t0, 72(a0)
        sd t1, 80(a0)
        sd t2, 88(a0)
        sd s0, 96(a0)
        sd s1, 104(a0)
        sd a1, 120(a0)
        sd a2, 128(a0)
        sd a3, 136(a0)
        sd a4, 144(a0)
        sd a5, 152(a0)
        sd a6, 160(a0)
        sd a7, 168(a0)
        sd s2, 176(a0)
        sd s3, 184(a0)
        sd s4, 192(a0)
        sd s5, 200(a0)
        sd s6, 208(a0)
        sd s7, 216(a0)
        sd s8, 224(a0)
        sd s9, 232(a0)
        sd s10, 240(a0)
        sd s11, 248(a0)
        sd t3, 256(a0)
        sd t4, 264(a0)
        sd t5, 272(a0)
        sd t6, 280(a0)

    # 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
```

- What is happening?
  - Actually, I have no Idea What is going on.
    - But thanks to the detailed description and additional research, I barely understood What is happening.

1. What are these registers and instructions?
   - `sscratch` - According to RISC-V ISA, `sstatus` is a CSR, (control and status register) that can be handled in supervisior mode.
   - `csrr {reg} {CSR}` - CSR read. Load CSR's data to register.
   - `csrw {CSR} {reg}` - CSR write. Store register's data to CSR.
   - `csrrw {reg} {CSR} {reg}` - Swap atomic. I don't know about the details but I can understand It can make swapping more faster and safer.
2. What is trapframe? - I think, trapframe is "address space" that can save current process's execution data before executing another jobs. And Then, We jump to `usertrap` in `trap.c`

`kernel/trap.c`

Now we are in the beginning of the handler. in `usertrap(void)`, we check if it is a trap from User, and then set up next PC to the next instruction from `ecall`. Finally, we call the `syscall()` function.

```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.
```

```c
  w_stvec((uint64)kernelvec);

  struct proc *p = myproc();

  // save user program counter.
  p->trapframe->epc = r_sepc();

  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();
  } 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();
}
```

`kernel/syscall.c`

Finally, We arrived syscall.c. In the `syscall(void)` function, we can see lots of syscalls are mapped in Array, So when we set current process's a0, it returns a return value of indicated system call. `p->trapframe->a0 = syscalls[num]();` And then, After `usertrapret` and `userret`, next instruction is executed.

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

  num = p->trapframe->a7;
  // We stored the system call number in a7, in usys.S which is generated by usys.pl
  // and then, in trampoline.S, uservec saved current user program's registers to the trapframe.
  // and then, It jumps to usertrap in trap.c, and checks if the cause of the trap is a system call
(scause == 8).
  // If it is, it calls syscall() in syscall.c.
```

```c
  // So naturally, when we look at current trapframe's a7, we can get the system call number.
  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;
  }
}
```

Design

Process, is a Data structure, And by `fork()`, child process is copied from its own parent process. So I think there will be some information about parent process in process's data structure. Here is an definition of xv6 process Data Structure.

`proc.h`

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

  // p->lock must be held when using these:
  enum procstate state;
  void *chan;
  int killed;
  int xstate;
  int pid;                     // Process ID

  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process (What we want)

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;
  uint64 sz;
  pagetable_t pagetable;
  struct trapframe *trapframe;
  struct context context;
  struct file *ofile[NOFILE];
  struct inode *cwd;
  char name[16];
};
```

- Luckily, I can found a pointer which indicates its own parent process.
- So I think, if I access to current process's `&parent -> pid`, that's what I want!

Implementation

Add a System call, `getppid()`

I made a new file `mysyscall.c`.

```c
#include "types.h"
#include "riscv.h"
```

```c
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "spinlock.h"
#include "proc.h"

int
getppid(void)
{
    return myproc()->parent->pid;
}
```

I think It will work, So I Added the source in make file, Declared in `defs.h`, Implemented a wrapper function. in `mysyscall.c`

```c
int
sys_getppid(void)

{
    return getppid();
}
```

Make a user program, `ppid.c`

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

int
main(int argc, char *argv[])
{
    printf("My student ID is 2019092824\n");
    fprintf(1, "My pid is %d\n", getpid());
    fprintf(1, "My ppid is %d\n", getppid());
    exit(0);
}
```

Its a simple program that simply returns pid and ppid.

Result

It seems it works!

Trouble Shooting

Here's two problems.

1. While I build, I missed some header files.

I implemeted my system call referencing `sysproc.c`. So first time I implemented, I did not included some header files.

```c
#include "types.h"
#include "riscv.h"
#include "defs.h"
#include "proc.h"

// I use myproc, so mabye It will work only including proc.h
int
getppid(void)
{
    return myproc()->parent->pid;
}
```

But while making, It gave me some errors. So I changed my code like this.

```c
#include "types.h"
#include "riscv.h"
#include "defs.h"
#include "proc.h"
#include "param.h"
#include "memlayout.h"
#include "spinlock.h"

// I Should implement all the things that sysproc.c did.
int
getppid(void)
{
    return myproc()->parent->pid;
}
```

And it solved a problem,

2. Is it secure to just access parent?

In `proc.h`, there is a comment of parent pointer.

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

Okay, So what is `wait_lock`? I have no idea what is `wait_lock`, So I found in `proc.c`.

in `fork()`, There was actual usage of `wait_lock`.

```c
// Something before...

// helps ensure that wakeups of wait()ing
// parents are not lost. helps obey the
// memory model when using p->parent.
// must be acquired before any p->lock.
struct spinlock wait_lock;

int
fork(void){
  int i, pid;
  struct proc *np;
  struct proc *p = myproc();

  // Something before..
```

```
  acquire(&wait_lock);
  np->parent = p;
  release(&wait_lock);

  // Something after..
}
//Something after...
```

I can see that before accessing parent, we acquire and release a `wait_lock`. And inside of `proc.c`, it says you MUST acquire that thing.

OK. So I have to fix my code. But first, How can I access to the `wait_lock`? its declared on `proc.c`, and I have to use THAT `wait_lock`.

So here is my new implemetation.

```
#include "types.h"
#include "riscv.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "spinlock.h"
#include "proc.h"

extern struct spinlock wait_lock; // wait_lock is located in proc.c

int
getppid(void)
{
    struct proc *p = myproc(); // Get the current process
    int ppid;

    acquire(&wait_lock);       // Acquire the wait_lock to safely access p->parent
    ppid = p->parent->pid;     // Retrieve the parent process's PID
    release(&wait_lock);       // Release the lock after accessing p->parent

    return ppid;               // Return the parent process's PID
}
```

And What should this program do if there are no parent process? I think the possibility is scarce. because all of the process's are the child process of root(init) process. But if some other program terminated it's parent, problem will occur.

So I implemented some error detection.

```
#include "types.h"
#include "riscv.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "spinlock.h"
#include "proc.h"

extern struct spinlock wait_lock; // wait_lock is located in proc.c

int
getppid(void)
```

```c
{
    struct proc *p = myproc(); // Get the current process
    int ppid;

    acquire(&wait_lock);        // Acquire the wait_lock to safely access p->parent
    if (p->parent) {
        ppid = p->parent->pid; // Retrieve the parent process's PID
    } else {
        ppid = -1;             // No parent (shouldn't happen for normal processes)
    }
    release(&wait_lock);        // Release the lock after accessing p->parent
    return ppid;               // Return the parent process's PID
}
```

naturally, I have to make some changes on my User program.

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

int
main(int argc, char *argv[])
{
    printf("My student ID is 2019092824\n");
    fprintf(1, "My pid is %d\n", getpid());
    if(getppid() == -1){
        fprintf(1, "No parent process\n");
    } else {
        fprintf(1, "My ppid is %d\n", getppid());
    }
    exit(0);
}
```

Here is the result...

```
xv6 kernel is booting

init: starting sh
$ ppid
My student ID is 2019092824
My pid is 3
My ppid is 2
$
```

No problem, But the speed of program slightly decreased.