

系统调用与进程

概述

目标

1. 了解 RISC-v 架构的 trap 机制
2. 理解系统调用的实现机制
3. 熟悉添加新系统调用的过程
4. 掌握使用系统调用编写程序

简述

系统调用是操作系统实现各种高级功能的基础。系统调用提供了一组编程接口，使得应用程序能够访问操作系统提供的各种服务和资源，例如文件操作、网络通信、进程管理等。在实现系统调用时，通常会使用硬件所提供的机制，保存当前运行现场，并从用户态切换到内核态。

进程是操作系统中最为重要的资源管理单元。每个进程都有自己的地址空间、代码、数据、堆栈等资源，它们共享系统资源，但是相互独立。操作系统为每个进程都在内核中维护了一个数据结构，称之为 PCB（进程控制块）。进程是操作系统进行资源控制的基本单位，操控其对应的 PCB 就操控了整个进程。

系统调用和进程是操作系统中最为基本的部分，也是操作系统实现各种高级功能的基础。学习掌握这两项内容，是深入理解操作系统的第一步。

在本部分中，会介绍 XV6 系统是如何使用 RISC-V 平台提供的 trap 机制，实现系统调用的。在实验部分，会由浅入深的安排若干实验，从学习如何使用系统调用编写用户程序，到为 XV6 增添新的系统调用。

讲解

Trap

在 RISC-V 上有三种事件会导致 CPU 搁置普通指令的执行，并强制将控制权转移到处理该事件的特殊代码上。

- 系统调用：当用户程序执行 `ecall` 指令
- 异常：（用户或内核）指令做了一些非法的事情，例如除以零或使用无效的虚拟地址
- 设备中断：一个设备触发了中断使得当前程序运行需要响应内核设备驱动

RISC-V 使用陷阱（trap）作为这些情况的通用术语。

当处于用户态时，trap 会跳到 S-mode（内核态）。

当处于 S-mode 时，trap 会跳到 M-mode 或依然留在 S-mode。

关于Trap，我们有几个基本的目标

- 安全和隔离：
- 我们不想让用户代码介入到这里的 user/kernel 切换，否则有可能会破坏安全性。所以这意味着，trap 中涉及到的硬件和内核机制不能依赖任何来自用户空间东西。
- 对用户透明：
- 我们希望内核能够响应中断，之后在用户程序完全无感知的情况下再恢复用户代码的执行。

因此，我们至少要做：

- 需要保存32个用户寄存器。因为寄存器表明程序执行的状态。但是这些寄存器又要被内核代码所使用，所以在 trap 之前，你必须**先在某处保存这32个用户寄存器**。
- **程序计数器 PC 也需要在某个地方保存**，它几乎跟一个用户寄存器的地位是一样的，我们需要能够在用户程序运行中断的位置继续执行用户程序。
- 需要将 mode 改成 supervisor mode，因为我们想要使用内核中的各种各样的特权指令。
- **SATP** 寄存器现在正指向 user page table，而 user page table 只包含了用户程序所需要的内存地址空间映射，它并没有包含整个内核数据的地址映射。
- 所以在运行内核代码之前，我们需要将 **SATP** 寄存器指向 kernel page table。
 - 即需要从用户进程页表切换到内核页表
- 需要将栈寄存器 **sp** 指向位于内核的一个地址，因为我们需要一个栈来调用内核的 C 函数（所谓的内核栈）。

Trap之后，系统进入S-mode（内核态），与用户态相比，进入内核态后能做的事：

- 可以读写 CSR 控制寄存器了。
- 比如说，当你在 supervisor mode 时，你可以：读写 SATP 寄存器，也就是 page table 的指针；STVEC，也就是处理 trap 的内核指令地址；SEPC，保存当发生 trap 时的程序计数器；SSCRATCH 等等。
 - 在 supervisor mode 你可以读写这些寄存器，而用户代码不能做这样的操作。
- S-mode 下可以使用 PTE_U 标志位为0的 PTE。
- 当 PTE_U 标志位为1的时候，表明用户代码可以使用这个页表；如果这个标志位为0，则只有 supervisor mode 可以使用这个页表。
 - 需要特别指出的是，supervisor mode 中的代码并不能读写任意物理地址。
- 在 supervisor mode 中，就像普通的用户代码一样，也需要通过 page table 来访问内存。
 - 如果一个虚拟地址并不在当前由 SATP 指向的 page table 中，又或者 SATP 指向的 page table 中 PTE_U=1，那么 supervisor mode 不能使用那个地址。
 - 所以，即使我们在 supervisor mode，我们还是受限于当前 page table 设置的虚拟地址空间。
 - 当然，以上只是理论上的限制，在 XV6 系统中，我们把整个内存地址空间都映射到了内核页表里，即在 S-mode 下可以读写任何内存地址。

系统调用实现

接下来，我们以在用户态执行 `write()` 系统调用为例，说明整个 trap 的过程。

调用流程图如下（注：有（）的是C代码，没有的是汇编）：

首先，我们在用户态编写如下的C代码（比如 `sh.c` 中）

```
write(_fd,_ptr,5);
```

接着由编译器翻译成汇编代码

```
...
li a7,16 # SYS_write == 16
ecall
```

每一个系统调用都有一个调用号，定义在 `kernel/syscall.h` 中，把这个号存在 `a7` 寄存器中（根据 calling convention），然后调用 `ecall` 指令（`ecall` 是一条硬件指令，而不是一个函数）。

我们编译用户程序使用的是 `riscv64-linux-gnu-gcc`，可 `gcc` 是如何知道我们自己定义的系统调用号的呢？

这个问题可以在 `Makefile` 里找到答案。

libc of xv6

<https://linux.die.net/man/7/libc>

glibc 是 linux 系统中最底层的 api，几乎其它任何运行库都会依赖于 glibc。glibc 除了封装 linux 操作系统所提供的系统服务外，它本身也提供了许多其它一些必要功能服务的实现。

接下来的过程，其实就是在构建 xv6 上的 libc。

`Makefile` 如下

```
ULIB = $U/ulib.o $U/usys.o $U/printf.o $U/umalloc.o

_%: %.o $(ULIB)
    $(LD) $(LDFLAGS) -T $U/user.ld -o $@ $^      #step 3

$U/usys.S : $U/usys.pl
    perl $U/usys.pl > $U/usys.S                  #step 1

$U/usys.o : $U/usys.S
    $(CC) $(CFLAGS) -c -o $U/usys.o $U/usys.S    #step 2
```

以编译 `cat` 用户程序为例

```
perl user/usys.pl > user/usys.S

riscv64-linux-gnu-gcc -c -o user/usys.o user/usys.S

riscv64-linux-gnu-ld -o user/_cat user/cat.o user/ulib.o user/usys.o
user/printf.o user/umalloc.o
```

首先使用 `perl` 从 `usys.pl` 生成汇编代码 `usys.S`，你可以理解为 `perl` 根据规则，自动生成了一些字符串。

生成的 `usys.S` 如下，注意第一行，这就是 `gcc` 知道 `SYS_write == 16` 的原因。我们在 `kernel/syscall.h` 定义了我们自己系统调用的系统调用号。

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

...

.global write
write:
    li a7, SYS_write
    ecall
    ret
```

接着用 `gcc` 根据 `usys.S` 生成 `usys.o`，之后每个用户程序都会链接上这个 `usys.o`。

由此生成的二进制文件，就可以在我们的 xv6 上运行了。

其实这一整个过程就是在构建 xv6 上的 libc。libc 除了封装 linux 操作系统所提供的系统服务外，它本身也提供了许多其它一些必要功能服务的实现。libc 里只是对系统调用进行 C 语言的封装，ta 并不关心 syscall 具体是如何实现的。

回到主题，将 `SYS_write` 放到 `a7` 寄存器后，便执行了 `ecall`，执行 `ecall` 指令后，便进入了 S-mode。

ecall(environment call)，当我们在 U 态执行这条指令时，会触发一个 ecall-from-u-mode-exception，从而进入 S 模式中的中断处理流程（常用来进行系统调用）。

RISC-V 硬件对所有陷阱（Trap）类型（计时器中断除外）执行以下操作：

1. 如果陷阱是设备中断，并且状态**SIE**位被清空，则不执行以下任何操作。
 1. **SIE** 寄存器，用来表示在 S-mode 下是否响应中断
2. 清除**SIE**以禁用中断。
3. 将 **pc** 复制到 **sepc**。
4. 将当前模式（U-mode或S-mode）保存在状态的**SPP**位中。
5. 设置 **scause** 以反映产生 Trap 的原因。
6. 将模式设置为 S-mode。
7. 将 **stvec** 复制到 **pc**。
8. 在新的 **pc** 上开始执行。

简单说 **ecall** 做了这几件事

- 从 U-mode 进入 S-mode
- 关中断
- 把 **pc** 存到 **sepc** 寄存器
- 把 **stvec** 的值给 **pc** 寄存器

除此之外，**ecall** 没做任何事情，没有切换page table，没有保存/修改其他寄存器。

也正是因为 **ecall** 没有切换 page table（也就是说现在虽然在 S-mode 下，但地址空间还是用户的地址空间），但改变了 **pc** 寄存器，因此想要能继续正确执行代码，处理 Trap 的代码必须在用户的地址空间里能够访问到。

所以这意味着，trap 处理代码必须存在于每一个 user page table 中。因为 ecall 并不会切换 page table，我们需要在 user page table 中的某个地方来执行最初的内核代码。而这个 trampoline page，是由内核小心地映射到每一个 user page table 中，以使得当我们仍然在使用 user page table 时，内核在一个地方能够执行 trap 机制的最开始的一些指令。

下图是用户的地址空间，用户地址空间顶部有 **trampoline**，其映射着处理 trap 的代码。

执行完 **ecall** 后，**pc** 寄存器为 **0x3fffffff000**，也就是在 trampoline page 的最开始。

我们现在正在 trampoline page 中执行程序，这个 page 包含了内核的 trap 处理代码。

内核已经事先设置好了 **stvec** 寄存器的内容为 **0x3fffffff000**，这就是 trampoline page 的起始位置。**stvec** 保存着**中断服务程序的入口地址**，也就是 **0x3fffffff000**，在执行 **ecall** 时，由硬件将该值赋给 **pc** 寄存器。

那么，**stvec** 寄存器是什么时候被赋值成 **0x3fffffff000** 也就是 trampoline 的地址的呢？

XV6根据执行的是用户代码还是内核代码对 **stvec** 有特别的处理，如下：

- 当在用户态时，**stvec** 指向 **uservec** (kernel/trampoline.S)
- 每次从内核态进入用户态，会执行 **usertrapret()** 函数，在这个函数中，会 **w_stvec(trampoline_uservec)**
- 当在内核态时，**stvec** 指向 **kernelvec** (kernel/kernelvec.S)
- 从用户态进入内核态时，会执行 **usertrap()** 函数，在这个函数里，会执行 **w_stvec((uint64)kernelvec)**

回到 `write()` 系统调用的流程，在执行完 `ecall` 后，进入了内核态，并跳到 `uservec` 执行。

`uservec`

该函数位于 `kernel/trampoline.S`，由汇编代码编写。

这个函数基本就干了两件事：

- 保存寄存器
- 加载寄存器

保存寄存器：

因为进入了内核态，所以可以读写用户地址空间中的 `trapframe` 段了。

`trapframe` 的地址位于 `sscratch` 寄存器中，同样的每次从内核态进入用户态都会设置 `sscratch` 为 `trapframe` 的地址。我们把所有寄存器保存在 `trapframe` 中，之后就可以自由使用寄存器了。

`trapframe` 虽然映射在用户地址空间，但在页表里设置了其访问权限，其只能在 S-mode/M-mode 下访问，在用户态下无法访问。

为什么这些寄存器保存在 `trapframe`，而不是用户代码的栈中？

这个问题的答案是，我们不确定用户程序是否有栈，必然有一些编程语言没有栈，对于这些编程语言的程序，Stack Pointer 不指向任何地址。当然，也有一些编程语言有栈，但是或许它的格式很奇怪，内核并不能理解。比如，编程语言以堆中以小块来分配栈，编程语言的运行时知道如何使用这些小块的内存在来作为栈，但是内核并不知道。所以，如果我们想要运行任意编程语言实现的用户程序，内核就不能假设用户内存的哪部分可以访问，哪部分有效，哪部分存在。所以内核需要自己管理这些寄存器的保存，这就是为什么内核将这些内容保存在属于内核内存的 `trapframe` 中，而不是用户内存。

这就是所谓的保存用户进程上下文（context）。实际上，寄存器就是上下文，保存寄存器就是保存上下文，恢复寄存器就是恢复上下文。

加载寄存器：

- 从进程的 `trapframe` 中加载 `sp` 寄存器，将栈切换到内核栈。
- 加载 `satp` 寄存器，切换页表，从用户地址空间切换到内核地址空间。

这里还有个问题，为什么代码没有崩溃？毕竟我们在内存中的某个位置执行代码，`pc` 保存的是虚拟地址，如果我们切换了 page table，为什么同一个虚拟地址不会通过新的 page table 寻址走到一些无关的 page 中？看起来我们现在没有崩溃并且还在执行这些指令。有人来猜一下原因吗？

我不知道你们是否还记得 user page table 的内容，trampoline page 在 user page table 中的映射与 kernel page table 中的映射是完全一样的。

这两个 page table 中其他所有的映射都是不同的，只有 trampoline page 的映射是一样的，因此我们在切换 page table 时，寻址的结果不会改变，我们实际上就可以继续在同一个代码序列中执行程序而不崩溃。这是 trampoline page 的特殊之处，它同时在 user page table 和 kernel page table 都有相同的映射关系。

之所以叫 trampoline page，是因为你某种程度在它上面“弹跳”了一下，然后从用户空间走到了内核空间。

然后跳到 `usertrap()` 执行。

usertrap()

因为切换到了内核栈，现在就可以执行 C 代码了。

```
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(x: (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();
    }
```

上文讲过，当在内核态时，`stvec` 指向 `kernelvec` (kernel/kernelvec.S)，首先要来修改这个寄存器，因为当在内核态和用户态的 trap 处理程序不同。

然后通过 `scause` 寄存器判断 trap 来源。

- The Interrupt bit in the `scause` register is set if the trap was caused by an interrupt.
- The Exception Code field contains a code identifying the last exception or interrupt.

Interrupt	Exception Code	Description	
1	0	<i>Reserved</i>	
1	1	Supervisor software interrupt	
1	2-4	<i>Reserved</i>	
1	5	Supervisor timer interrupt	
1	6-8	<i>Reserved</i>	
1	9	Supervisor external interrupt	
1	10-15	<i>Reserved</i>	
1	≥ 16	<i>Designated for platform use</i>	
0	0	Instruction address misaligned	
0	1	Instruction access fault	
0	2	Illegal instruction	
0	3	Breakpoint	
0	4	Load address misaligned	
0	5	Load access fault	
0	6	Store/AMO address misaligned	
0	7	Store/AMO access fault	
0	8	Environment call (ecall) from U-mode (即系统调用)	
0	9	Environment call (ecall) from S-mode	

因为我们是执行 `write` 系统调用，而进入 Trap，因此 `scause` 寄存器里的值为 8.

`if (r_scause() == 8)` 通过这条语句的判断，我们就被导向了实现系统调用的代码。

然后 `p->trapframe->epc += 4`。因为是系统调用（ecall），我们希望在下一条指令恢复，也就是 ecall 之后的一条指令。所以对于系统调用，我们对于保存的用户程序计数器加4，这样我们会在 ecall 的下一条指令恢复，而不是重新执行 ecall 指令。

然后 `intr_on()`，中断总是会被 RISC-V 的 trap 硬件流程所关闭，所以在这个时间点，我们需要显式的打开中断。

然后执行 `syscall()`

`syscall()`，真正执行系统调用的地方


```

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,
    ....[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,
};

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 {
        ..printfk("%d %s: unknown sys call %d\n", p->pid, p->name, num);
        ..p->trapframe->a0 = -1;
    }
}

```

它的作用是从 `syscalls` 表中，根据系统调用的编号查找相应的系统调用函数。

如果你还记得之前的内容，Shell 调用的 `write` 函数将 `a7` 寄存器设置成了系统调用编号，对于 `write` 系统调用来说就是16。

所以 `syscall` 函数的工作就是获取由 `trampoline` 代码保存在 `trapframe` 中 `a7` 寄存器的数字，然后用这个数字索引，去调用真正的系统调用。

`p->trapframe->a0 = syscalls[num]();` 这条语句就是真正实现系统调用的地方，调用对应的系统调用的具体实现函数，然后将返回值保存在 `a0` 寄存器中。

对于 `write` 系统调用，就是执行 `syscalls[16]()` 函数，也就是 `sys_write(void)` 函数。

`syscalls` 是一个函数数组，其保存着各个系统调用真正实现的函数地址，系统调用号就是这个数组的下标。

`sys_write(void)`


```
uint64 sys_write(void) { You, 3周前
..struct file *f;
..int n;
..uint64 p;

..argaddr(1, &p);
..argint(2, &n);
..if (argfd(n: 0, pfd: 0, pf: &f) < 0)
...return -1;

..return fwrite(f, p, n);
}
```

这里就是 `write` 系统调用实现的地方，首先从用户空间获得系统调用的参数，然后去执行真正的实现。

关于 `fwrite()` 具体是如何实现的，我们并不关心，我们目前只关系，系统调用实现的流程。

返回 `usertrap()`

执行完 `fwrite`，便一路 `return`，最终回到 `usertrap()`。

`fwrite()` <-- `sys_write()` <-- `usertrap()`

在 `usertrap()` 函数的最后，调用了 `usertrapret()`。

```
...intr_on();

...syscall();
/..} else if ((which_dev = devintr()) != 0) {
...// ok
/..} else {
...printfk("usertrap(): unexpected scause %p pid=%d\
...printfk(".....sepc=%p stval=%p\n", r_sepc(
...setkilled(p);
..}

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

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

..usertrapret(); You, 3周前 • first commit
}
```

然后调用流程进入 `usertrapret()`

`usertrapret()`

这个函数里，我们首先关闭了中断。

我们之前在系统调用的过程中是打开了中断的，这里关闭中断是因为我们将要更新 `STVEC` 寄存器来指向用户空间的 trap 处理代码，而之前在内核中的时候，我们指向的是内核空间的 trap 处理代码。

我们关闭中断因为当我们将 `STVEC` 更新到指向用户空间的 trap 处理代码时，我们仍然在内核中执行代码。如果这时发生了一个中断，那么程序执行会走向用户空间的 trap 处理代码，即便我们现在仍然在内核中，出于各种各样具体细节的原因，这会导致内核出错。

所以我们这里关闭中断。

在下一行我们设置了 `STVEC` 寄存器指向 `trampoline` 代码，在那里最终会执行 `sret` 指令返回到用户空间。

位于 `trampoline` 代码最后的 `sret` 指令会重新打开中断。

这样，即使我们刚刚关闭了中断，当我们在执行用户代码时中断是打开的。

```
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(x: trampoline_uservec);

    // set up trapframe values that uservec will need when
    // the process next traps into the kernel.
    p->trapframe->kernel_satp = r_satp(); // kernel page table
    p->trapframe->kernel_sp = p->kstack + PGSIZE; // process's kernel stack
    p->trapframe->kernel_trap = (uint64)usertrap;
    p->trapframe->kernel_hartid = r_tp(); // hartid for cpuid()
}
```

接下来的几行填入了 `trapframe` 的内容，这些内容对于执行 `trampoline` 代码非常有用。这里的代码就是：

- 存储了 kernel page table 的指针
- 存储了当前用户进程的 kernel stack
- 存储了 usertrap 函数的指针，这样 trampoline 代码才能跳转到这个函数（注，详见6.5中 `ld t0 (16)a0` 指令）
- 从 tp 寄存器中读取当前的 CPU 核编号，并存储在 trapframe 中，这样 trampoline 代码才能恢复这个数字，因为用户代码可能会修改这个数字

现在我们在 `usertrapret` 函数中，我们正在设置 `trapframe` 中的数据，这样下一次从用户空间转换到内核空间时可以用到这些数据。

```

..// tell trampoline.S the user page table to switch to.
..uint64 satp = MAKE_SATP(p->pagetable);

..// jump to userret in trampoline.S at the top of memory, which
..// switches to the user page table, restores user registers,
..// and switches to user mode with sret.
..uint64 trampoline_userret = TRAMPOLINE + (userret - trampoline);
..((void (*)(uint64))trampoline_userret)(satp);
}

```

然后进入汇编代码 `userret`

`userret`

此时，又回到了汇编。

`userret` 接受一个参数 `pagetable`，根据 Calling-Convention，这个参数放在 `a0` 寄存器里。

切换到用户页表，恢复原来用户的寄存器，回到 U-mode.

```

.....ld s8, 224(a0)
.....ld s9, 232(a0)
.....ld s10, 240(a0)
.....ld s11, 248(a0)
.....ld t3, 256(a0)
.....ld t4, 264(a0)
.....ld t5, 272(a0)
.....ld t6, 280(a0)

→      # restore user a0
.....ld a0, 112(a0)
.....

.....# return to user mode and user pc.
.....# usertrapret() set up sstatus and sepc.
.....sret

```

`sret` 是我们在 kernel 中的最后一条指令，当执行完这条指令：

- 程序会切换回 user mode
- SEPC 寄存器的数值会被拷贝到 PC 寄存器（程序计数器）
- 重新打开中断（这一点是由 `sret` 硬件指令完成的）

至此，`write` 系统调用执行完毕，从 `write` 系统调用的下条指令继续执行。

```

write(_fd, _ptr, 5);
// 从系统调用返回，接着执行

```

XV6 的 PCB（进程控制块）

进程控制块（Process Control Block，PCB）是操作系统内核中用于管理进程的重要数据结构。每个进程都有一个对应的 PCB，用于存储和管理该进程的所有信息。

下图是 XV6 进程 PCB 的定义。

```
// Per-process state
You, 2周前 | 7 authors (Robert Morris and others)
struct proc {      rsc, 17年前 • more comments ...
..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;.....// switch() here to run process
..struct file *ofile[NOFILE];..// Open files
..struct inode *cwd;.....// Current directory
..char name[16];.....// Process name (debugging)

..int trace_mask;.....// for sys_trace
};
```

这是一个简化的 xv6 中 struct proc 结构体定义，它包含了管理进程所需的所有信息。具体各个字段的含义如下：

- `struct spinlock lock`：自旋锁，用于保护该进程的 PCB 不被并发访问和修改。
- `enum procstate state`：进程状态，取值为就绪（RUNNABLE）、阻塞（SLEEPING）或运行（RUNNING）等。
- `void *chan`：如果进程处于睡眠状态，则该字段保存了进程正在等待的事件。
- `int killed`：如果该字段的值为非零，则表示该进程已经被杀死。
- `int xstate`：进程退出时返回给父进程的状态信息。
- `int pid`：进程 ID，用于唯一标识该进程。
- `struct proc *parent`：父进程的指针。
- `uint64 kstack`：该进程在内核态使用的堆栈地址。
- `uint64 sz`：进程占用的内存大小。
- `pagetable_t pagetable`：进程的页表，用于管理进程的虚拟内存地址空间。
- `struct trapframe *trapframe`：中断帧，用于进程从用户态切换到内核态时保存 CPU 寄存器的值。
- `struct context context`：用于保存进程的 CPU 上下文信息，包括进程寄存器的值和栈指针等。
- `struct file *ofile[NOFILE]`：打开的文件列表，用于管理进程打开的文件。
- `struct inode *cwd`：当前工作目录的 inode。
- `char name[16]`：进程名字，用于调试和输出日志等。

这些字段共同组成了 xv6 中进程的 PCB，通过访问和修改这些字段，内核可以实现对进程的管理和调度。

为 XV6 添加新的系统调用

在本小部分，我演示如何为 XV6 添加一个新的系统调用 `trace`。

`trace` 系统调用来控制跟踪。它应该有一个参数，这个参数是一个整数“掩码”（mask），它的比特位指定要跟踪的系统调用。

例如，要跟踪 `fork` 系统调用，程序调用 `trace(1 << SYS_fork)`，其中 `SYS_fork` 是 `kernel/syscall.h` 中的系统调用编号。

如果在掩码中设置了系统调用的编号，则必须修改 xv6 内核，以便在每个系统调用即将返回时打印出一行。该行应该包含进程 id、系统调用的名称和返回值。

`trace` 系统调用应启用对调用它的进程及其随后派生的任何子进程的跟踪，但不应影响其他进程。

以下是两个示例：

```
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 966
3: syscall read -> 70
3: syscall read -> 0

$ trace 2147483647 grep hello README
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 966
4: syscall read -> 70
4: syscall read -> 0
4: syscall close -> 0
```

在上面的第一个例子中，`trace` 调用 `grep`，仅跟踪了 `read` 系统调用。32 是 `1<<SYS_read`。

在第二个示例中，`trace` 在运行 `grep` 时跟踪所有系统调用；2147483647 将所有31个低位置为1，即跟踪所有系统调用。

步骤：

- 在 `kernel/syscall.h` 里，为 `trace` 系统调用分配一个系统调用号

```
h syscall.h M ×
kernel > h syscall.h
You, 50秒钟前 | 3 authors (Frans Kaashoe
1 // System call numbers
2 #define SYS_fork....1
3 #define SYS_exit....2
4 #define SYS_wait....3
5 #define SYS_pipe....4
6 #define SYS_read....5
7 #define SYS_kill....6
8 #define SYS_exec....7
9 #define SYS_fstat...8
10 #define SYS_chdir...9
11 #define SYS_dup....10
12 #define SYS_getpid 11
13 #define SYS_sbrk...12
14 #define SYS_sleep..13
15 #define SYS_uptime 14
16 #define SYS_open...15
17 #define SYS_write..16
18 #define SYS_mknod..17
19 #define SYS_unlink 18
20 #define SYS_link...19
21 #define SYS_mkdir..20
22 #define SYS_close..21
23 |
24 #define SYS_trace..22
25 |
```

这里，我们就将 `SYS_trace` 定义为 22，为 `trace` 系统调用分配系统调用号。

- 实现 `trace` 函数
- 在 `kernel/sysproc.c` 中添加一个 `sys_trace()` 函数。

```
kernel > C sysproc.c > sys_trace
97
98 uint64
99 sys_trace(void) You, 2周前 • syscall
100 {
101     ..argint(0, &(myproc()->trace_mask));
102     ..return 0;
103 }
104
```

可以看到，这里仅仅是将 `trace` 系统调用的参数保存到了当前进程 PCB 的 `trace_mask` 字段。

进程 PCB 的 `trace_mask` 字段本来是不存在的, 为了实现 `trace` 系统调用, 同时也要修改 xv6 进程 PCB 的定义。

```
You, 2周前 | 7 authors (Robert Morris and others)
v 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
    ..int killed;.....// If non-zero, have been k
    ..int xstate;.....// Exit status to be return
    ..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
    ..uint64 kstack;.....// Virtual address of kerne
    ..uint64 sz;.....// Size of process memory (
    ..pagetable_t pagetable;.....// User page table
    ..struct trapframe *trapframe; // data page for trampoline
    ..struct context context;.....// swtch() here to run proc
    ..struct file *ofile[NOFILE];..// Open files
    ..struct inode *cwd;.....// Current directory
    ..char name[16];.....// Process name (debugging)

    ..int trace_mask;.....// for sys_trace
};
rtm, 17年前 • import ...
```

然后, 修改 `kernel/syscall.c` 中的 `syscall()` 函数, 以真正实现 `trace` 函数, 在进行对应的系统调用时, 打印输出。

```
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]();
    ..if ((1 << num) & p->trace_mask)
    ..    printf("%d: syscall %s -> %d\n", p->pid, syscalls_name[num], p->trapframe->a0);
    ..} else {
    ..    printf("%d %s: unknown sys call %d\n",
    ..        p->pid, p->name, num);
    ..    p->trapframe->a0 = -1;
    ..}
}
```


这里的修改并不是添加一个系统调用的必须步骤，而是 `trace` 系统调用的实现机制。


- 在 `kernel/syscall.c` 里，声明 `sys_trace` 函数，然后将 `sys_trace()` 函数添加系统调用数组 `syscalls` 里

```
kernel > C syscall.c > ...
91 extern uint64 sys_chdir(void);
92 extern uint64 sys_dup(void);
93 extern uint64 sys_getpid(void);
94 extern uint64 sys_sbrk(void);
95 extern uint64 sys_sleep(void);
96 extern uint64 sys_uptime(void);
97 extern uint64 sys_open(void);
98 extern uint64 sys_write(void);
99 extern uint64 sys_mknod(void);
100 extern uint64 sys_unlink(void);
101 extern uint64 sys_link(void);
102 extern uint64 sys_mkdir(void);
103 extern uint64 sys_close(void);
104 extern uint64 sys_trace(void);
105
```



```
// 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,
    [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_trace]..= sys_trace,
};
```



You, 16分钟前 • Uncommi

当用户程序执行系统调用时，根据系统调用号（`SYS_trace`），在系统调用数组（`syscalls[]`）里查找，就能找到对应系统调用的真正实现了。

其实至此，一个新的系统调用已经添加到内核里了。

但用户程序还用不了这个新的系统调用，因为还没有更新我们 Xv6 对应的 `libc`。

- 在 `user/user.h` 里，添加 `trace` 系统调用的声明

```
user > h user.h > trace
3 // system calls
4 int fork(void);
5 int exit(int) __attribute__((noret
6 int wait(int*);
7 int pipe(int*);
8 int write(int, const void*, int);
9 int read(int, void*, int);
10 int close(int);
11 int kill(int);
12 int exec(const char*, char**);
13 int open(const char*, int);
14 int mknod(const char*, short, shor
15 int unlink(const char*);
16 int fstat(int fd, struct stat*);
17 int link(const char*, const char*)
18 int mkdir(const char*);
19 int chdir(const char*);
20 int dup(int);
21 int getpid(void);
22 char* sbrk(int);
23 int sleep(int);
24 int uptime(void);
25 int trace(int);
26
```

我们新添加的 `trace` 系统调用，接受一个 `int` 类型的参数。

- 在 `user/usys.pl` 里，为 `trace` 系统调用添加新的存根

```

user >  usys.pl
16     }
17     →
18     entry("fork");
19     entry("exit");
20     entry("wait");
21     entry("pipe");
22     entry("read");
23     entry("write");
24     entry("close");
25     entry("kill");
26     entry("exec");
27     entry("open");
28     entry("mknod");
29     entry("unlink");
30     entry("fstat");
31     entry("link");
32     entry("mkdir");
33     entry("chdir");
34     entry("dup");
35     entry("getpid");
36     entry("sbrk");
37     entry("sleep");
38     entry("uptime");
39     entry("trace");

```

一个新的系统调用 `trace` 就添加完成了，我们可以在用户程序中使用。

新建一个文件 `user/trace.c`，编写一个用户程序，就可以正常使用 `trace` 系统调用了。

```

#include "kernel/param.h"
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int
main(int argc, char *argv[])
{
    int i;
    char *nargv[MAXARG];

    if(argc < 3 || (argv[1][0] < '0' || argv[1][0] > '9')){
        fprintf(2, "Usage: %s mask command\n", argv[0]);
        exit(1);
    }

    if (trace(atoi(argv[1])) < 0) {
        fprintf(2, "%s: trace failed\n", argv[0]);
    }
}

```

```

    exit(1);
}

for(i = 2; i < argc && i < MAXARG; i++){
    nargv[i-2] = argv[i];
}
exec(nargv[0], nargv);
exit(0);
}


```

- 在 **Makefile** 里，为 `UPROGS` 添加新的用户程序 `_trace`

```

174  UPROGS=\
175  →      $U/_cat\
176  →      $U/_echo\
177  →      $U/_forktest\
178  →      $U/_grep\
179  →      $U/_init\
180  →      $U/_kill\
181  →      $U/_ln\
182  →      $U/_ls\
183  →      $U/_mkdir\
184  →      $U/_rm\
185  →      $U/_sh\
186  →      $U/_stressfs\
187  →      $U/_usertests\
188  →      $U/_grind\
189  →      $U/_wc\
190  →      $U/_zombie\
191  →      $U/_trace\
192  |

```



XV6 只是一个操作系统内核，内核本身不包含用户程序。

而我们直接接触使用的软件其实是众多用户程序，只不过这些用户程序用到了内核的系统调用。

这一步是为 XV6 添加新的用户程序，类似于在系统上安装一个新软件。

运行起 XV6，执行用户程序 `trace`

```
gemu-system-riscv64 -machine virt -bios none -kern
,format=raw,id=x0 -device virtio-blk-device,drive=

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 961
3: syscall read -> 321
3: syscall read -> 0
$ |
```

操作系统内核与用户程序

上文提到

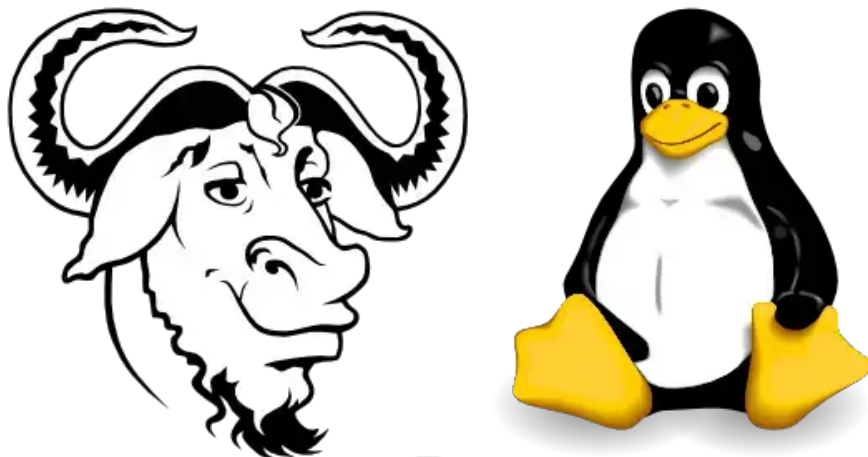
在 `kernel/syscall.c` 里，将 `sys_trace()` 函数添加系统调用数组 `syscalls` 里

在这步之后，其实就已经将新的系统调用 `trace()` 添加到操作系统内核里了。但此时在用户程序还无法使用，因为还没有更新 XV6 系统对应的 `libc`。

我们平时所使用的操作系统，从我们用户的角度，我们认为它运行起来之后，就会有可视化的 GUI 界面，或者至少有一个终端，可以给 Shell 输入一些命令。

但从开发者角度，广义上的操作系统其实是分为两大部分的：内核 & 用户程序。实际这两部分的开发维护更新过程，也是两波人在分别负责。

以 Linux 操作系统为例，通常我们只是称其为 Linux，但其完整的称呼应该是 GNU/Linux。



GNU/Linux

这里的 GNU 指众多用户程序的集合；Linux 指 Linux 操作系统内核。

许多用户并不了解作为内核的 Linux 和也被称作“Linux”的整个系统的区别。而不加区别地使用该名称并不能帮助人们对此的理解。这些用户常常认为 Linus Torvalds 在 1991 年凭借一些帮助完成了整个操作系统的开发。

程序员一般知道 Linux 是一个内核。但是他们一般也听到整个系统叫“Linux”，他们通常会设想的历史是整个

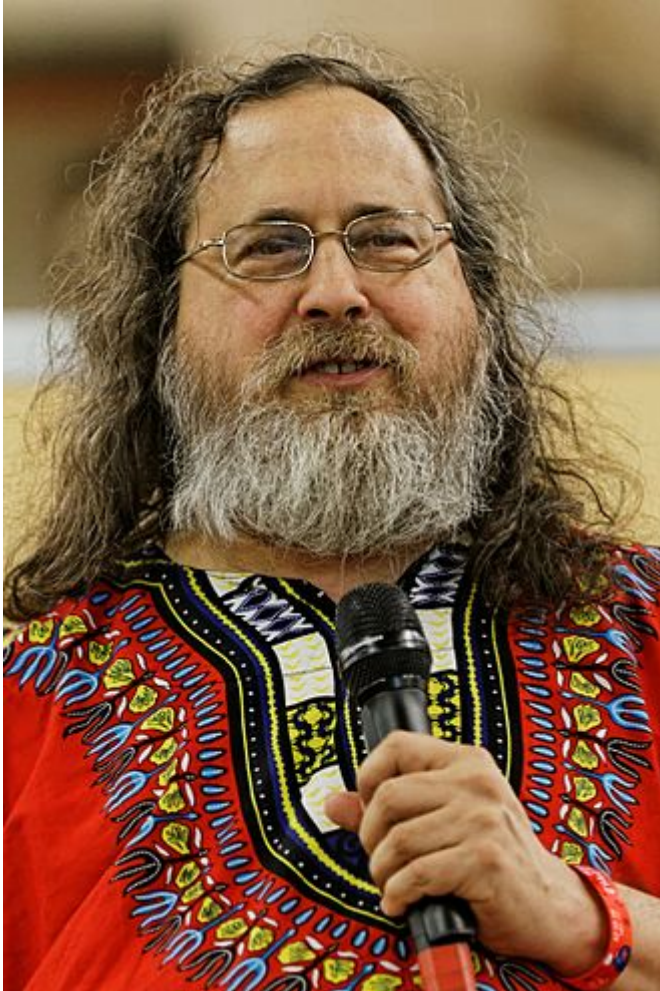
系统要以内核命名。例如，许多人相信一旦 Linux Torvalds 完成了 Linux 内核，其用户就搜索一些自由软件来和内核一起工作，此时他们发现（没有特别的原因）构成一个类似 Unix 系统的大多数必要组件已经有了（也就是 GNU）。

<https://www.gnu.org/gnu/linux-and-gnu.html>

有个计划叫 GNU 计划，它的愿景就是开发开源的操作系统-GNU，虽然开发了操作系统之上的大量工具，但核心的操作系统内核一直没有开发出来。后来发现 Linux 内核可以直接使用，就拿来组合成了 GNU/Linux 操作系统，也就是我们后来的各大 Linux 发行版。

<https://www.zhihu.com/question/319783573>

GNU 里的历史（1983）比 Linux 内核（1991）更久远，其创始人是 RMS（理查德·斯托曼），也就是下图这个大胡子。

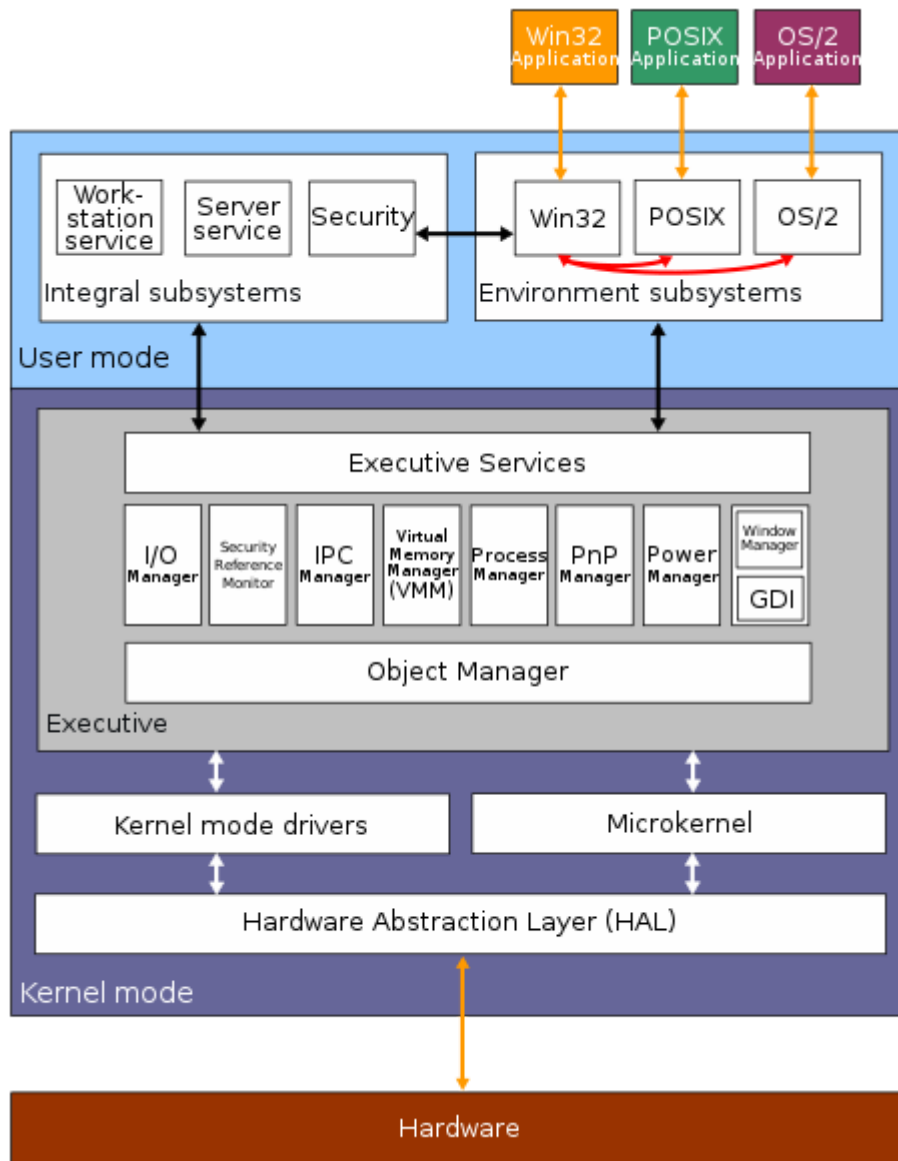


RMS 发起了 GNU 项目，并领导了自由软件运动。我们所熟知的 `gcc` 编译器，`bash` 等都是 GNU 的一部分。

操作系统内核本身不包含任何用户能直接用到的东西，用户直接接触到的都是用户程序。

这一点并不是十分严谨的说法。在 Windows 系统上，窗口管理器是内核的一部分，地位比较高。而在 Linux 上，窗口管理器只是一个普通的用户程序，你可以自由的更换，如 Kde，GNOME 等等

下图是 Windows NT 内核的架构图，仅供了解。



实验

实验介绍

1. 编写若干用户程序，熟悉如何使用系统调用
2. 添加新的系统调用 `getppid`，返回父进程的 pid
3. 添加新的系统调用 `getschedtime`，显示进程最近一次被调度运行的开始与结束时间，通过参数传回用户空间。

实验内容 1 - Hello World

```
> git add .
> git commit -m 'message' #暂存修改
> git switch lab2-1-hello #切换到本实验的分支
```

任务：

- 实现 xv6 的用户程序 `hello`：
- `hello` 程序仅仅简单打印字符串 `Hello World` 到终端。

- 另一种说法是，输出字符串 `Hello World` 到标准输出。
- 解决方案应该放在 **user/hello.c**

要求：

- 不能使用 `printf`，`puts`，`putchar` 函数
- 为了美观，请输出换行符

Tips：

- `write()` 是一个系统调用，用于向文件描述符（file descriptor，简称为 fd）指定的文件、管道、套接字等输出数据。
- 在 Unix 和类 Unix 系统中，0、1和2是预留给标准输入、标准输出和标准错误的文件描述符（fd）。

预期输出：

```
$ make qemu
...
init: starting sh
$ ./hello
Hello World
$
```

```
qemu-system-riscv64 -machine virt -bios
,format=raw,id=x0 -device virtio-blk-dev

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ ./hello
Hello World
$ |
```

实验内容 2 - find

```
> git add .
> git commit -m 'message' #暂存修改
> git switch lab2-2-find #切换到本实验的分支
```

任务：

- 实现 xv6 的用户程序 `find`：
- `find` 程序，接收两个参数（搜索路径和目标文件名称），查找目录树中具有特定名称的所有文件
 - 解决方案应该放在 **user/find.c**

`find` 是一个 Linux/Unix 操作系统中常用的命令行工具，用于在指定目录下搜索文件或目录。它可以根据不同的搜索条件进行匹配，并将查找到的结果输出到标准输出中。`find` 命令还可以通过执行指定的操作来处理搜索结果，如删除、复制、移动等。`find` 命令的基本语法是：

```
find [path] [expression]
```

其中 `[path]` 为要查找的目录路径，`[expression]` 为查找条件，可以是文件名等等。

例如，要在当前目录及其子目录下查找所有文件名为 `hello` 的文件，可以使用以下命令：

```
find . hello
```

这条命令会从当前目录开始递归查找所有所有文件名为 `hello` 的文件，并将结果输出到标准输出中。

Tips:

- 在 C 语言中，我们可以使用 `main` 函数来接受命令行参数。
- `main` 函数有两个参数：`argc` 和 `argv[]`。其中，`argc` 表示传递给程序的参数数量，包括程序本身，而 `argv[]` 表示一个指针数组，每个元素存储一个参数的值，其中第一个元素 `argv[0]` 存储的是程序名，后面的元素 `argv[1]`、`argv[2]` 等存储的是传递给程序的其他 d 参数。
 - https://www.gnu.org/software/libc/manual/html_node/Program-Arguments.html
- 查看 `user/lsc` 文件学习如何读取目录
- 可以使用递归下降，遍历子目录
- 不要在“.”和“..”目录中递归
- 对文件系统的更改会在 qemu 的运行过程中一直保持；要获得一个干净的文件系统，请运行 `make clean`，然后 `make qemu`
- 你将会使用到 C 语言的字符串，要学习它请看《C 程序设计语言》(K&R)，例如第 5.5 节
- 注意在 C 语言中不能像 python 一样使用“==”对字符串进行比较，而应当使用 `strcmp()`
- 将程序加入到 `Makefile` 的 `UPROGS`

预期输出：

```
$ make qemu
...
init: starting sh
$ echo 1 > b
$ mkdir a
$ echo 1 > a/b
$ find . b
./b
./a/b
$
```

```

qemu-system-riscv64 -machine virt
,format=raw,id=x0 -device virtio

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ echo 1 > b
$ mkdir a
$ echo 1 > a/b
$ find . b
./b
./a/b
$ |
$ |

```

实验内容 3 - 添加系统调用 `getppid`

```

> git add .
> git commit -m 'message'    #暂存修改
> git switch lab2-3-getppid  #切换到本实验的分支

```

任务：

- 添加系统调用 `getppid`，以使用户程序可以获取其父进程的 `pid`。
- 函数声明为 `int getppid(void);`

Tips:

- 为了添加 `getppid` 系统调用，我们需要以下几个步骤：

1. 在 `syscall.h` 文件中定义系统调用号
2. 在 `sysproc.c` 文件中添加系统调用的实现代码，并编写对应的内核函数
3. 在 `syscall.c` 文件中添加系统调用的声明，并添加到系统调用数组里
4. 在 `user.h` 文件中声明用户程序可以调用的函数
5. 在 `usys.pl` 文件中为新系统调用添加新的存根
6. 在用户程序中调用 `getppid` 系统调用，获取父进程的 `pid`

- XV6 进程 PCB 里 `parent` 字段指向其父进程的 PCB
- 每个进程 PCB 里 `pid` 字段为其自己的 `pid`
- 每个进程都有父进程，唯一特别的是 `init` 进程的父进程还是他自己。

预期输出：

```

$ make qemu
...
init: starting sh
$ ./getppidtest
pid: 4, ppid: 3
pid: 3, ppid: 2
$

```

```
qemu-system-riscv64 -machine
,format=raw,id=x0 -device vi

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ ./getppidtest
pid: 4, ppid: 3
pid: 3, ppid: 2
$ |
```

实验内容 4 - 添加系统调用 `getschedtime`

```
> git add .
> git commit -m 'message'    #暂存修改
> git switch lab2-4-getsched  #切换到本实验的分支
```

任务：

- 添加系统调用 `getschedtime`，获取进程最近一次被调度运行的开始与结束时间，通过参数传递，将结果从内核空间拷贝到用户空间。
- 函数声明为

```
struct schedtime {
    int stime; // start time of ticks
    int etime; // end time of ticks
};
int getschedtime(struct schedtime*);
```

一个滴答(tick)是由 `xv6` 内核定义的时间概念，即来自定时器芯片的两个中断之间的时间。

假如进程是第一次被调度，则其 `stime` 非零，为其被调度运行的 ticks 数；其 `etime` 为零，因为该进程还没有结束过一次调度时间片。

Tips：

- 参照添加 `getppid` 系统调用的过程
- 在 `XV6` 进程 PCB 里添加必要的字段，在每次进程被调度切换时，更新维护该字段
- 系统调用时，从进程的 PCB 对应字段获得数据，拷贝到用户空间。
- 可以参照 `kernel/sysfile.c` 的 `sys_pipe` 函数，学习如何从用户空间获得系统调用参数，如何在内核地址空间与用户地址空间之间拷贝数据。

预期输出：

```
qemu-system-riscv64 -machine virt -bios none  
,format=raw,id=x0 -device virtio-blk-device,  
  
xv6 kernel is booting  
  
init: starting sh  
$ ./getschedtimetest  
child last sched start at 67, end at 0  
parent last sched start at 68, end at 69  
$ |
```

maybe code bug here

拓展实验内容

编写一个 Shell 以执行用户指令，要求支持基本的 `|` 管道操作（e.g `cat a.txt | wc -l`）。

Tips:

- <https://www.cs.purdue.edu/homes/grr/SystemsProgrammingBook/Book/Chapter5-WritingYourOwnShell.pdf>
- <https://blog.ehoneahobed.com/building-a-simple-shell-in-c-part-1>