XV6

thebesttv

July 8, 2022

Contents

- xv6 book rev11
- Xv6 for RISC-V git repo
- 6.S081: Operating System Engineering - Lab guides
- xv6 general information
- Homework: running and debugging xv6
- xv6-explained Explanations of xv6 operating system

File descriptor convention

- 0: stdint
- 1: stdout
- 2: stderr

Two file descriptors **share an offset** if they were derived from the same original file descriptor by a sequence of `fork()` and `dup()` calls.

The shell command `2>&1` tells shell to make fd 2 a duplicate of 1.

> Xv6 is written in "LP64" C, which means long (L) and pointers (P) in the C programming language are 64 bits, but int is 32-bit.
>
> Xv6 is written for the support hardware simulated by qemu's `-machine virt` option. This includes RAM, a ROM containing boot code, a serial connection to the user's keyboard/screen, and a disk for storage.

'virt' Generic Virtual Platform (`virt`) - QEMU

Only when `PTE_U` is set can a user process access that page when running in user mode.

Scheduler

- round-robin
- size of timeslice is fixed using `int interval = 1000000;` in `timerinit()` (in `kernel/start.c`)
- all cores share one "ready queue"
- the next timeslice for process may be running on a different core

`gcc` warns about infinite recursion in `user/sh.c:runcmd()`, to solve this, either

- ignore the warning by adding `-Wno-infinite-recursion` to `CFLAGS` in makefile
- or declare the function does not return using `__attribute__((noreturn))`

```
// Execute cmd.  Never returns.
__attribute__((noreturn))
void
runcmd(struct cmd *cmd)
{
```

1 Locks

used to protect shared data

1. init lock
2. acquire lock
3. critical section
4. release lock

1.1 Spin lock

If interrupt is not disabled between `acquire()` and `release()`, possible situation of deadlock

- one process holds the lock
- interrupt
- handler tries to acquire the same lock

1.2 Sleep lock

2 QEMU

Use `C-a c` to go to monitor/console. Use `info mem` to show the memory pages.

```
QEMU 7.0.0 monitor - type 'help' for more information
(qemu) info mem
vaddr            paddr            size             attr
---------------- ---------------- ---------------- -------
0000000000000000 0000000087f61000 0000000000001000 rwxu-a-
0000000000001000 0000000087f5e000 0000000000001000 rwxu-a-
0000000000002000 0000000087f5d000 0000000000001000 rwx----
0000000000003000 0000000087f5c000 0000000000001000 rwxu-ad
0000003fffffe000 0000000087f70000 0000000000001000 rw---ad
0000003ffffff000 000000008000a000 0000000000001000 r-x--a-
```

3 GDB

3.1 Break to certain global address

In `user/sh.asm`, the `write` function is at 0x14fc

```
00000000000014fc <write>:
.global write
write:
 li a7, SYS_write
    14fc: 48c1                li a7,16
 ecall
    14fe: 00000073           ecall
```

```
 ret
    1502: 8082                ret
```

To break at that address, add `*` before the address

```
(gdb) b *0x14fc
Breakpoint 1 at 0x14fc
(gdb) c
Continuing.

Breakpoint 1, 0x00000000000014fc in ?? ()
=> 0x00000000000014fc:  c1 48   li      a7,16
```

Print `pc` for the current address and display arguments passed to `write`. So the original call was `write(2, ">", 1)`.

```
(gdb) print $pc
$1 = (void (*)()) 0x14fc
(gdb) i r a0 a1 a2
a0             0x2      2
a1             0x3ecb   16075
a2             0x1      1
(gdb) x/1c $a1
0x3ecb: 62 '>'
```

4 Devices

- UART
- disk
- timer interrupt: local to each hart
- PLIC (Platform-Level Interrupt Controller)
- CLINT (Core Local Interruptor): local to each hart

5 Lecture 3: OS Organization and System Calls

ecall instruction: transfer control to kernel `ecall syscall-number`

6 Lecture 4: Page Tables (VM)

`kalloc`

```
// Allocate one 4096-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
```

reg `satp` points to physical addr of page table.

- each process has its own addr space
- each CPU core has its own `satp` (of course)
- changes on each context switch
- kernel stores each process's addr table (value of `satp`)

Sv39

- 4KB page (12 bits offset)
- physical addr has 56 bits in total, $56 - 12 = 44$ bits PFN
- virtual addr only has 39 bits (2^{39} B = 512 GB) in total, $39 - 12 = 27$ bits page index (VPN)
- each page stores $4096/8 = 512$ entries
- 3-level page table
  - L2 $\rightarrow$ L1 $\rightarrow$ L0 $\rightarrow$ PPN + offset
  - $9 + 9 + 9 + 12 = 39$
- `satp` & all entries in PTEs (i.e., all page directories) are **physical addr**

Hardware looks up PTE, but we have a `walk` function implemented by OS. The hardware lookup is used in load/store instructions for address in the current page table, while `walk` is used when visiting addresses from **another page table**. Xv6 has a separate page table for each process and **one for the kernel address space**. So the total number of page tables are process count + 1. `walk` is used in `walkaddr`, which, in-turn, is used by `copyinstr` (in `kernel/vm.c`) for copying string from user space to kernel space on a syscall.

TLB—cache of PTEs

- TLB needs to be flashed on every context switch
  - `sfence.vla` does this

caches

- some indexed by virtual addr (before MMU)
- others by physical addr (after MMU)

Two central functions in `kernel/vm.c`:

- `walk` finds the PTE of a virtual address in the specified page table
- `mappages` installs PTEs for new mappints

`main`

- `kvminit` creates kernel page table → `kvmmake` → `kvmmap`
- `kvminithart` installs page table

`proc_pagetable` creates a user page table for a given process. It installs translation for the trampoline page, the trapframe, and, in the page table lab, the USYSCALL page. It assumes the trapframe & the USYSCALL page are already allocated and stored in PCB (in `p->trapframe` and `p->usyscall`). It does no memory allocation except for creating the page table. It is referenced by

- `exec` to replace the existing process with a new image. Note here that the process is originally **created** by fork, so the trapframe and the USYSCALL page have already been allocated and stored in PCB.
- `allocproc` which creates a new process. It, in turn, is called by
  - `userinit` to set up the first user process
  - `fork` to create a new process by copying the parent

The creation and destruction of a process are controlled by

- `allocproc` which
  - finds an unused PCB entry
  - allocate a new PID, trapframe, and USYSCALL page
  - creates a new page table and installs translations by calling `proc_pagetable`

- `freeproc`
  - frees trapframe, USYSCALL page
  - frees the pagetable by calling `proc_freepagetable`
  - resets the PCB entry to unused

Ways to create a new process

- `exec`
- `fork`

Note: if excuted successfully, `allocproc` does not release the lock of the PCB entry. The entry is returned and its lock is later released in `userinit` and `fork`, the two functions that creates a new process.

6.1 Lab page table

6.1.1 Speed up system calls

Only three places need to be modified

- page allocation & deallocation
  - in `allocproc`, allocate the USYSCALL page with `kalloc` (before calling `proc_pagetable`)
  - in `freeproc`, use `kfree` to free the allocated USYSCALL page
- page table mapping & unmapping
  - in `proc_pagetable`, install mapping with `mappages` (VA `TRAPFRAME` to PA `p->trapframe`)
  - in `proc_freepagetable`, unmap the page with `uvmunmap` (not sure whether it's necessary)
- store address of the USYSCALL page in PCB by adding pointer to `usyscall`

6.1.2 Print a page table

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
page table 0x0000000087f6e000
 ..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
 .. ..0: pte 0x0000000021fda401 pa 0x0000000087f69000
 .. .. ..0: pte 0x0000000021fdac1f pa 0x0000000087f6b000 urwx   # code
 .. .. ..1: pte 0x0000000021fda00f pa 0x0000000087f68000 -rwx   # guard page
 .. .. ..2: pte 0x0000000021fd9c1f pa 0x0000000087f67000 urwx   # stack
 ..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
 .. ..511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
 .. .. ..509: pte 0x0000000021fdd813 pa 0x0000000087f76000 ur-- # USYSCALL
 .. .. ..510: pte 0x0000000021fddc07 pa 0x0000000087f77000 -rw- # trapframe
 .. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000 -r-x # trampoline
```

6.2 Kernel Memory Layout

`kernel/memlayout.h` declares the constants for the kernel memory layout, such as `PHYSTOP` and `KERNBASE`. Note that `MAXVA` is declared in `kernel/riscv.h`.

Stack grows downward

Each process has its own kernel stack. The kernel stack maps high in memory (right below the trampoline page), with a gurad page below it to prevent overflow.

7 Lecture 6: Isolation & system call entry/exit

Privileged CPU features

- read & write privileged registers
  - `satp`: physical address of page table root
  - `stvec`: ecall jumps here, points to trap handler in trampoline page
  - `sepc`: ecall saves the user's `pc` here
  - `sscratch`: scratch space; used to store temporary data (mainly used to save general registers to the trap frame)
- privileged instructions

  - Access regs: `csrr` (read), `csrw` (write), `csrrw` (swap)
  - `sret`: return to userspace

- use PTEs that don't have `PTE_U` set
- still uses page table

Three kinds of *trap* that force a transfer of control to handler code

- system call: a user executes the `ecall` **instruction**
- exception: an **instruction** does something illegal, such as divide by zero or use an invalid virtual address
- interrupt: when a **device** signals it needs attention

  The SPP bit indicates the privilege level at which a hart was executing before entering supervisor mode. When a trap is taken, SPP is set to 0 if the trap originated from user mode, or 1 otherwise. When an SRET instruction (see Section 3.3.2) is executed to return from the trap handler, the privilege level is set to user mode if the SPP bit is 0, or supervisor mode if the SPP bit is 1; SPP is then set to 0.

  The SIE bit enables or disables all **interrupts** in supervisor mode. When SIE is clear, interrupts are not taken while in supervisor mode. When the hart is running in user-mode, the value in SIE is ignored, and supervisor-level interrupts are enabled. The supervisor can disable individual interrupt sources using the `sie` CSR.

  The SPIE bit indicates whether supervisor interrupts were enabled **prior to trapping into supervisor mode**. When a trap is taken into supervisor mode, SPIE is set to SIE, and SIE is set to 0. When an SRET instruction is executed, SIE is set to SPIE, then SPIE is set to 1.

The process of the `write()` syscall:

- The program calls `write(1, "Hello, World\n", 13);`. According to the RISC-V calling convention, the first argument (`1`) is placed in `a0`, the address of the second (`"Hello, World\n"`) in `a1`, and the third (`13`) in `a2`. After all the arguments are placed in registers, the assembled code jumps to `write`.

```
write(1, "Hello, World\n", 13);
12: 4635                li a2,13
14: 00001597           auipc a1,0x1
18: d7c58593           addi a1,a1,-644 # d90 <malloc+0x142>
1c: 4505                li a0,1
1e: 00000097           auipc ra,0x0
22: 536080e7           jalr 1334(ra) # 554 <write>
```

- The wrapper function `write` is defined in `user/usys.S`, which is generated by `user/usys.pl`. The function simply places the syscall num (`SYS_write`, defined in `kernel/syscall.h`), calls `ecall`, and then returns.

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

- The `ecall` instruction causes a trap. To allow maximum flexiblity, the RISC-V hardware does **minimal preparation**, including
  - disabling interrupts by clearing the SIE bit in `sstatus`
  - saving `pc` to `sepc`
  - saving the current mode in the SPP bit in `sstatus`
  - setting `scause` to reflect the trap's cause
  - switching to supervisior mode
  - jumping to `stvec` (setting `pc` to `stvec`)

  Note that the hardware does not switch to kernel page table, set up kernel stack, or preserve registers. All these are done in software.

- `stvec` is set to `uservec()` (in `kernel/trampoline.S`), which is in the trampoline page. It is the first function called in kernel mode. It saves all registers to the trapframe, switches to kernel stack and kernel page table, and jumps to `usertrap()`.

- `usertrap()` inspects the `scause` register and sees that the trap is caused by an environment call from U-mode (a syscall), updates `epc` to point to the next instruction (`epc+=4`), and then lets `syscall()` handle it.

- `syscall()` retrieves the syscall num in `a7` from the trapframe (`p->trapframe->a7`), and calls the corresponding function `sys_write()` (in `kernel/sysfile.c`) that does the real work.
- In the actual syscall function such as `sys_write`, the original arguments (`a0`, `a1`, `a2`) are now in the trapframe.
- After `sys_write()` returns, `syscall()` places the return value in `p->trapframe->a0`.
- After `syscall()` returns to `usertrap()`, the latter calls `usertrapret()`.
- `usertrapret()` updates the trapframe, and finally calls `userret()`.
- `userret()` is also in the trampoline page. It switches to user page table, restores the registers from trapframe, and uses `sret` to return to user mode.
- `sret` returns to user mode and re-enables interrupt.

7.1 Slight modification to the original `userret()`

The original `userret()` (in `kernel/trampoline.S`) first sets `sscratch` to the original `a0`, then restores all registers but `a0`, and finally swap `a0` with `sscratch` so `a0` has now its original value, and `sscratch` the `TRAPFRAME`.

Another solution is to first restore all registers but `a0`, set `sscratch` to `a0` (TRAPFRAME), and finally restore `a0` from the trapframe. No swapping is needed.

```
.globl userret
userret:
        # userret(TRAPFRAME, pagetable)
        # switch from kernel to user.
        # usertrapret() calls here.
        # a0: TRAPFRAME, in user page table.
        # a1: user page table, for satp.

        # switch to the user page table.
        csrw satp, a1
        sfence.vma zero, zero

        # restore all but a0 from TRAPFRAME
        ld ra, 40(a0)
        ld sp, 48(a0)
```

```
ld gp, 56(a0)
ld tp, 64(a0)
ld t0, 72(a0)
ld t1, 80(a0)
ld t2, 88(a0)
ld s0, 96(a0)
ld s1, 104(a0)
ld a1, 120(a0)
ld a2, 128(a0)
ld a3, 136(a0)
ld a4, 144(a0)
ld a5, 152(a0)
ld a6, 160(a0)
ld a7, 168(a0)
ld s2, 176(a0)
ld s3, 184(a0)
ld s4, 192(a0)
ld s5, 200(a0)
ld s6, 208(a0)
ld s7, 216(a0)
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)

csrw sscratch, a0       # set sscratch to TRAPFRAME
ld a0, 112(a0)          # restore a0

# now all 31 registers have their original value
# and sscratch has TRAPFRAME

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

8 Lecture 7: Page Faults

Implementing VM features using page faults

- lazy allocation
- copy-on-write fork
- demand paging
- mma

VM provides

- isolation
- a lavel of indirection
  - trampoline page
  - guard page

Using page faults, we can change mapping dynamically on-the-fly.
On a page fault, information needed

- the faulting va (in `stval`)
- the type of fault (in `scause`)
  - load page fault (13)
  - store page fault (15)
  - instruction page fault (12)
- the va of instruction that caused the fault (in `p->trapframe->sepc`)

8.1 Lazy Allocation

An application tend to over-ask memory needed.

- `sbrk()` only updates `p->size`, do not really allocate memory
- on page fault
  - va $<$ `p->size`, allocate page, zero it, and insert mapping in page table
  - va $>=$ `p->size`, over the bounds

8.2 Zero-Fill on Demand

On program startup, many pages in PTEs are initialised to zeros. Map these all-zero pages to one read-only page. When one page is written, a page fault occurs, and the handler

- allocate a new page
- overwrite it with zeros
- update PTE
- re-execute the faulting instruction

8.3 Copy-on-Write Fork

When using `fork()` to create a child process, the child is a complete duplicate of its parent. Instead of allocating new pages and copying parent's memory, simply copy the parent's page table. The difference, however, is that all pages, **both in parent and child**, are now marked readonly. When the parent / child writes to a page, a page fault occurs, and the handler

- allocate a new page
- copy the parent's page content
- update PTE, marking the new page writable
- re-execute the faulting instruction

8.4 Demand Paging

8.5 Memory-Mapped Files

Load file contents into memory, so simple load/store instructions can read/write file.

- `mmap(va, len, fd, ...)` map file to memory
- `unmap(va, len, ...)` unmap

9 Lecture 8: Interrupts

Interrupt, different from syscall

- asynchronous
  - interrupt is triggered by external evenets, it has nothing to do with the current running process
  - syscall is triggered by instructions, it runs **in the context of the process**
- concurrency with CPU & IO
- driver

9.1 Where do interrupts come from

(focusing mostly on external interrupt, not software/timer interrupt)

Platform-Level Interrupt Controller (PLIC)

- manages interrupt from external devices
- route interrupts to certain cores

Driver manages devices; has two parts

- top part: user program consults
- bottom part: interrupt handler, does run in the context of a specific proces

when the UART finishes transmitting, it generates an interrupt to indicate that it can transmit another byte

what happens with `$ ls`

- `$` (uart output)
  - device puts `$` into uart
  - uart sends the char
  - uart generates interrupts when the char has been sent
- `ls` (keyboard input) `l` + `s` + `LF`
  - keyboard connect to the receive line

  - keyboard sends `l`
  - generates interrupt, interrupt handler

interrupt registers

- SIE: has one bit for each differnt traps (exception, software, timer)
- SSTATUS: one bit that globally enables/disables interrupt
- SIP: interrupt pending
- SCAUSE
- STVEC
- `main()`
  - `consoleinit()` (only on hart 0)
    - `uartinit()`
  - `plicinit()` (only on hart 0)
  - `plicinithart()` (on all harts)

interrupt enable & disable by SSTATUS

```
// enable device interrupts
static inline void intr_on() {
  w_sstatus(r_sstatus() | SSTATUS_SIE);
}
// disable device interrupts
static inline void intr_off() {
  w_sstatus(r_sstatus() & ~SSTATUS_SIE);
}
```

- `write()` syscall
- `sys_write()`
- `filewrite()`
- `consolewrite()`
- `uartputc()`

what hardware does on an interrupt (when SIE bit is set)

- clear SIE bit
- SEPC <- PC
- save current mode
- mode <- supervisior
- PC <- STVEC (jump to `usertrap()`)

10 Lab: traps

Lab: traps

10.1 RISC-V assembly

To prevent function inlining, use

```
int __attribute__ ((noinline)) g(int x) {
  return x+3;
}
int __attribute__ ((noinline)) f(int x) {
  return g(x);
}
```

11 Lab: cow

Notes

- COW needs `walk()` to get the PTE of the virtual address. However, unlike `walkaddr()`, `walk()` will panic when seeing invalid virtual address (one that's equal to or above `MAXVA`). Use a `safe_walk()` that ensures `va` to be below `MAXVA` before calling `walk()` (just like what `walkaddr()` does before calling `walk()`), while also checking that the PTE is user-accessible (also see `walkaddr()`) before returning it. Otherwise, return zero.
- When using multiple cores, synchronization becomes important. Pay very close attention to when and how `pagecount` needs a lock, especially in `cow_copy_page()`.

12 Calling Convention

soft-float convention for implementations lacking floating-point units (e.g., RV32I/RV64I)

Compressed format only uses 8 regs (`x8..15`), so `s0` and `s1` are searate from other `sN` regs. The same is true for `aN` regs.

return value: `a0` and `a1`, so when returning an object that's twice the size of the pointer word (word size), `a1` is used to hold the upper word.

12.1 Caller Saved vs Callee Saved

`ra` is caller saved—when doing function call, it's important for the callee to be able to modify `ra` if it wants to call another funciton.

12.2 Stack

Stack starts from high addr and grows **downwards** to low addr.

- `sp`: bottom of stack
- `fp`: top of current frame—return addr & prev frame pointer always at a fixed addr

A function call generates a **stack frame**.

- return addr
- previous frame pointer

Leaf function does not call other functions, so they don't need to save `ra`, etc.

ASM function structure

- prolog
- body
- epilog

13 GDB command

`x`

`watch`
`ptype`
`tui enable`, `layout split/reg/asm`
`watch`

14 Startup

14.1 `_entry` (in `entry.S`)

```
        # qemu -kernel loads the kernel at 0x80000000
        # and causes each CPU to jump there.
        # kernel.ld causes the following code to
        # be placed at 0x80000000.
.section .text
.global _entry
_entry:
        # set up a stack for C.
        # stack0 is declared in start.c,
        # with a 4096-byte stack per CPU.
        # sp = stack0 + (hartid * 4096)
        la sp, stack0
        li a0, 1024*4
        csrr a1, mhartid        # a1 = hart (core) id
        addi a1, a1, 1          # a1 = hartId + 1
        mul a0, a0, a1
        add sp, sp, a0          # sp = stack0 + 4096 * (heartId + 1)
        # jump to start() in start.c
        call start
```

Set up a stack for each core (hart). `stack0` is defined in `start.c`.

```
// entry.S needs one stack per CPU.
// NCPU is max number of CPUs, defined in param.h, default to 8
__attribute__ ((aligned (16))) char stack0[4096 * NCPU];
```

Each hart gets a 4096-byte stack. The stack grows down (what is down ???), so `sp` of hart 0 actually gets `stack0` + 4096. Here `csrr` (Control Status Register Read ???) reads the value of `mhartid` (Hart ID Register[1]) to `a1`. After stack is set, each core jumps to `start`.

14.2 `start` (in `start.c`)

The machine instruction `mret` switches to supervisor mode from machine mode. `mepc` is used to as "return" address.

Finally `start` "returns" to supervisor mode by calling `mret` and jumps to `main`.

[1] p.20, 3.1.5 Hart ID Register `mhartid`, *riscv-privileged-20211203.pdf*.

14.3 `main` (in `main.c`)

according to different hart id, initialize several devices & subsystems

calls `userinit` (in `proc.c`) to create the first process

the first process is an assembly code process, its machine code is in `initcode` (`proc.c:214`), original file `user/initcode.S`. This first process calls `exec` to replace itself with the program `/init`. Init (user/init.c:15) creates a new console device file if needed and then opens it as file descriptors 0, 1, and 2. Then it starts a shell on the console.

15 Syscall

`argint`, `argaddr` and `argstr` all use `argraw` to retrieve the nth arguments passed to system call as int, address, and string.

Syscalls show up in many places.

To see how syscall arguments are retrieved, refer to section 4.4 of the book.

15.1 In user

- `user/user.h` declares all syscalls for user processes to use
- `user/usys.pl` is a Perl script that generages `usys.S`, which is used as entry point of syscalls.

  On a syscall, such as `write`, the function jumps to asm code in `usys.S`

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

  The function simply puts the syscall number `SYS_write` (in `kernel/syscall.h`) in reg `a7` and calls `ecall`, transfering control to OS. When the OS finally completes the syscall and returns from `ecall`, `write` calls `ret` to return back to the calling process.

15.2 In kernel

- `kernel/syscall.h` defines all syscall numbers for `user/usys.S` and `kernel/syscall.c` to use

- `kernel/syscall.c` implements function `argraw`, `argint`, `argaddr` to retrieve arguemnt in syscall. It also implements `void syscall(void)` (at the end) as entry point for all the syscalls.
- `kernel/sysproc.c` actually implements some of the syscalls (some syscalls, such as `fstat`, are in `kernel/sysfile.c`). All syscalls are function in the form of `uint64 sys_xxx(void)`. They get their arguments through functions like `argint` and `argaddr` (in `syscall.c`).

15.3 About Lab Syscall

Finally, you can trace the suscall made by `sysinfotest`. As `write` is used to print to console, and the test calls `sbrk` to exhaust memory, leave `SYS_write` (16) and `SYS_sbrk` (12) out of mask. ($2^{31} - 1 - 2^{16} - 2^{12}$)

```
$ trace 2147414015 sysinfotest
3: syscall trace -> 0
3: syscall exec -> 1
sysinfotest: start
3: syscall sysinfo -> 0
3: syscall sysinfo -> -1
3: syscall sysinfo -> 0
3: syscall sysinfo -> 0
3: syscall sysinfo -> 0
3: syscall sysinfo -> 0
3: syscall sysinfo -> 0
3: syscall fork -> 4
4: syscall sysinfo -> 0
3: syscall wait -> 4
3: syscall sysinfo -> 0
sysinfotest: OK
```

16 Run XV6

Install risc-v toolchain & qemu:

```
sudo pacman -S riscv64-linux-gnu-binutils riscv64-linux-gnu-gcc riscv64-linux-gnu-gdb q
```

Clone the repo:

```
git clone git@github.com:mit-pdos/xv6-riscv.git
```

Then start:

```
make qemu
```

To quit QEMU, type `C-a x`.

17 Syscalls

```
int exit(int status)
int kill(int pid)
int getpid()
int sleep(int n)
int exec(char *file, char *argv[])
char *sbrk(int n)
int open(char *file, int flags)
int write(int fd, char *buf, int n)
int read(int fd, char *buf, int n)
int close(int fd)
int dup(int fd)
int pipe(int p[])
int chdir(char *dir)
int mkdir(char *dir)
int mknod(char *file, int, int)
int fstat(int fd, struct stat *st)
int stat(char *file, struct stat *st)
int link(char *file1, char *file2)
int unlink(char *file)
```

Terminate the current process; status reported to wait(). No return. Terminate process PID. Returns 0, or -1 for error. Return the current process's PID. Pause for n clock ticks. Load a file and execute it with arguments; only returns if error. Grow process's memory by n bytes. Returns start of new memory. Open a file; flags indicate read/write; returns an fd (file descriptor). Write n bytes from buf to file descriptor fd; returns n. Read n bytes into buf; returns number read; or 0 if end of file. Release open file fd. Return a new file descriptor referring to the same file as fd. Create a pipe, put read/write file descriptors in p[0] and p[1]. Change the current directory. Create a new directory. Create a device file. Place info about an open file into *st. Place info about a named file into *st. Create another name (file2) for the file file1. Remove a file.

17.1 fork

```
int fork()
```

Create a process, return child's PID.

> A process may create a new process using the `fork` system call. `Fork` gives the new process exactly the same memory contents (both instructions and data) as the calling process. `Fork` returns in both the original and new processes. In the original process, `fork` returns the new process's PID. In the new process, `fork` returns zero. The original and new processes are often called the *parent* and *child*.

On error, returns -1.

Although `fork()` copies the file descriptor table, each underlying file offset is shared between parent and child. For example,

```
if (fork() == 0) {
  write(1, "hello ", 6);
  exit(0);
} else {
  wait(0);                      // wait for child to finish
  write(1, "world\n", 6);
}
```

will always print `hello world\n`.

17.2 exit

> The `exit` system call causes the calling process to stop executing and to release resources such as memory and open files. `Exit` takes an integer status argument, conventionally 0 to indicate success and 1 to indicate failure.

17.3 wait

```
int wait(int *status);
```

Wait for a child to exit; exit status in *status; returns child PID.

> The wait system call returns the PID of an exited (or killed) child of the current process and copies the exit status of the child to the address passed to wait; if none of the caller's children has exited, wait waits for one to do so. If the caller has no children, wait immediately returns -1. If the parent doesn't care about the exit status of a child, it can pass a 0 address to wait.

17.4 kill

17.5 getpid

17.6 sleep

A tick is a notion of time defined by the xv6 kernel, namely the time between two interrupts from the timer chip.

17.7 exec

`exec()` replaces the calling process's memory but **preserves its file table**, which allows the shell to implement I/O redirection.

18 2

About hart (hardware thread), see this. Most new expensive chips have multiple hardware threads per core. Hardware Multi-threading: a Primer

Stack is fixed at one page. It does not grow. Guard frame used (¬U, i.e., not accessible in user mode)

Each process has its own trap frame & "trampoline" page, both ¬U. When an interrupt occurs, code in trampoline page saves the entire state of the process to its trap frame.

Arguments (argc, argv) are pushed on the one-page stack bofore program starts.

Sv39, suppoed 39 bits, but only 38 bits are used (MAXVA, `kernel/riscv.h:363`).

```
#define MAXVA (1L << (9 + 9 + 9 + 12 - 1))
```

19 3

entry.S (set up stack `sp`, `tp`) -> start.c -> main.c (each core will execute `main` in parallel)

param.h defs.h