دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

دانشکده مهندسی برق

گزارش‌کار پروژه سیستم عامل

Xv6

نگارش

علی بابالو

استاد راهنما

دکتر جوادی

آذر ماه 1401

abstract

xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix Version 6 (v6). xv6 loosely follows the structure and style of v6, but is implemented for a modern RISC-V multiprocessor using ANSI C.

Contents

Page #

Phase one

https://github.com/cavendishsama/xv6-OS.git

Xv6 boot steps:

In first step we explain every function which is called in main function until the first process is created.

```
volatile static int started = 0;

// start() jumps here in supervisor mode on all CPUs.
void
main()
{
  if(cpuid() == 0){
    consoleinit();
    printfinit();
    printf("\n");
    printf("xv6 kernel is booting\n");
    printf("\n");
    kinit();         // physical page allocator
    kvminit();       // create kernel page table
    kvminithart();   // turn on paging
    procinit();      // process table
    trapinit();      // trap vectors
    trapinithart();  // install kernel trap vector
    plicinit();      // set up interrupt controller
    plicinithart();  // ask PLIC for device interrupts
    binit();         // buffer cache
    iinit();         // inode table
    fileinit();      // file table
    virtio_disk_init(); // emulated hard disk
    userinit();      // first user process
    __sync_synchronize();
    started = 1;
  } else {
    while(started == 0)
      ;
    __sync_synchronize();
    printf("hart %d starting\n", cpuid());
    kvminithart();    // turn on paging
    trapinithart();   // install kernel trap vector
    plicinithart();   // ask PLIC for device interrupts
  }

  scheduler();
}

```

main(): This is the first function that is called. It initializes the console and printf, initializes the kernel memory allocator and page table, initializes the process table, initializes the trap vectors, sets up interrupt handling, initializes the buffer cache and inode table, initializes the emulated disk, and starts the first user process.

At the beginning main function check if cpu id is zero or not (r_tp function is called in cpuid which read the thread pointer which xv6 uses to hold this core's hartid / core's number, the index into CPUs[]). At the beginning the core number is zero because OS is not booted.

Xv6's main calls consoleinit (kernel/console.c:184) to initialize the UART hardware. This code configures the UART to generate a receive interrupt when the UART receives each byte of input, and a transmit complete interrupt each time the UART finishes sending a byte of output.

printfinit Initializes the console output buffer and sets up the console device for formatted output.

The function main calls kinit to initialize the allocator (kernel/kalloc.c:27). Kinit initializes the free list to hold every page between the end of the kernel and PHYSTOP. xv6 ought to determine how much physical memory is available by parsing configuration information provided by the hardware. Instead xv6 assumes that the machine has 128 megabytes of RAM. Kinit calls freerange to add memory to the free list via per-page calls to kfree. A PTE can only refer to a physical address that is aligned on a 4096-byte boundary (is a multiple of 4096), so freerange uses PGROUNDUP to ensure that it frees only aligned physical addresses. The allocator starts with no memory; these calls to kfree give it some to manage.

main calls kvminit, which Initializes the kernel's virtual memory system. This involves setting up the kernel page table, which maps the kernel's virtual address space to the physical memory, to create the kernel's page table. This call occurs before xv6 has enabled paging on the RISC-V, so addresses refer directly to physical memory. Kvminit first allocates a page of physical memory to hold the root page-table page.

After that kvminithart is called to install the kernel page table. It writes the physical address of the root page-table page into the register satp. After this the CPU will translate addresses using the kernel page table. Since the kernel uses an identity mapping, the now virtual address of the next instruction will map to the right physical memory address.

procinit, which is called from main, allocates a kernel stack for each process. It maps each stack at the virtual address generated by KSTACK, which leaves room for the invalid stack-guard pages. This function Initializes the process table used by the kernel to manage processes. It creates the first process, which is the idle process.

Then trapinit initializes the trap vector table used by the kernel to handle hardware exceptions, such as page faults or illegal instructions. This function sets up the table with default handlers for each type of exception. And then trapinithart sets up the trap vector table for the current CPU.

Plicinit initializes the Platform-Level Interrupt Controller (PLIC) used by the kernel to handle device interrupts. This function sets up the controller's registers and enables interrupts for each device. Plicinithart sets up the PLIC for the current CPU.

The buffer cache is a doubly linked list of buffers. The function binit called by main initializes the list with the NBUF buffers in the static array buf. All other access to the buffer cache refer to the linked list via bcache.head, not the buf array. This buffer cache is used to cache data read from and written to disk.

And iinit initializes the inode table used by the kernel to manage files.

Fileinit initializes the file table used by the kernel to manage open files.

virtio_disk_init initializes the emulated hard disk. For instance, If the buffer needs to be read from disk, bread calls virtio_disk_rw to do that before returning the buffer.

After main initializes several devices and subsystems, it creates the first process by calling userinit. The first process executes a small program written in RISC-V assembly, initcode.S, which re-enters the kernel by invoking the exec system call. As we saw in Chapter 1 of the book, exec replaces the memory and registers of the current process with a new program. Once the kernel has completed exec, it returns to user space in the /init process. Init 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. The system is up.

Compilers and CPUs follow rules when they re-order to ensure that they don't change the results of correctly written serial code. However, the rules do allow re-ordering that changes the results of concurrent code and can easily lead to incorrect behavior on multiprocessors. If such a re-ordering occurred, there would be a window during which another CPU could acquire the lock and observe the updated list but see an uninitialized list->next. To tell the hardware and compiler not to perform such re-orderings, xv6 uses __sync_synchronize in both acquire and release. __sync_synchronize is a memory barrier: it tells the compiler and CPU to not reorder loads or stores across the barrier. The barriers in xv6's acquire and release force order in almost all cases where it matters, since xv6 uses locks around accesses to shared data.

Started variable in main.c, used to prevent other CPUs from running until CPU zero has finished initializing xv6. started = 1 indicates that the kernel has finished booting and is now ready to run user processes.

After CPU zero has finished initializing the other CPUs also start to initialize.

After initializing, main function calls scheduler. Scheduler never returns, it loops, doing:

- Choose a process to run.
- Switch to start running that process.
- Eventually that process transfer control via switching back to the scheduler.

This loop continues until we shut down the OS.

Int getProcTick (int pid):

In this system call we will return the number of the ticks since a process is created in OS. For doing so we need some changes in OS files which will be explained below.

1- syscall.h

```
#define SYS_getProcTick 23
```

in this file we'll set the number corresponding to getProcTick system call.

2- sysproc.c

```
uint64
sys_getProcTick(void)
{
  int pid;

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

In this function we define ProcTick system call and call tickDiff function for return value. This means whenever this system call is called tickDiff function works.

3- proc.h

```
// 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
  int creation_time;           // ticks passed when a process is created
```

In this file we added creation_time variable to see how many ticks have passed since the start of the OS.

4- proc.c

```c
p->creation_time = ticks;
```

After an UNUSED process state turns to USED we define creation_time which is ticks passed since the beginning.

```c
int
tickDiff(int pid)
{
  struct proc *p;
  // int flag = 0;

  for(p = proc; p < &proc[NPROC]; p++){
    acquire(&p->lock);
    printf("%d: %s\n", p->pid, p->name);
    if (p->pid == pid){
      uint xticks = ticks;
      uint ProcTick;

      // acquire(&tickslock);
      // xticks = ticks;
      // release(&tickslock);

      ProcTick = xticks - p->creation_time;
      printf("ProcTick is equal to : %d\n", ProcTick);

      release(&p->lock);
      return 0;
    }

    release(&p->lock);
  }

  printf("There is no process with such pid");
  return 0;
}
```

In this function by iterating through all processes we seek to find the process with desired “pid” and by subtracting xticks from creation_time we will see the process was alive for this amount of ticks.

5- syscall.c

```c
extern uint64 sys_getProcTick(void);
```

```c
[SYS_getProcTick] sys_getProcTick,
```

By externing sys_getProcTick function, we’ll add this system call to OS’s function array pointer.

6- defs.h

```
int             tickDiff(int pid);
```

proc.c functions are accessible through this file and we must define tickDiff function for ProcTick system call her.

7- user.h

```
int getProcTick(int);
```

This file is an interface for users and all system calls are listed here.

8-usys.pl

```
entry("getProcTick");
```

9- ProcTickTest.c

```
xv6-riscv > user > C ProcTickTest.c > main(int, char **)
#include "./kernel/types.h"
#include "./kernel/stat.h"
#include "./user/user.h"

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

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

A test file for this system call.

10- Makefile

```
	$U/_ProcTickTest\
```

we change the make file UPORGS to see this system call and run it

Testing this system call:

```
$ ProcTickTest 1
1: init
ProcTick is equal to : 180
$ ProcTickTest 2
1: init
2: sh
ProcTick is equal to : 248
$
```

There’re only two processes and you can see they were alive for how many ticks.

Int sysinfo (struct sysinfo* info)

The process of defining the system call is explained before so we only explain functions.

1– sysinfo.h

xv6-riscv > kernel > C sysinfo.h > ...

```c
struct sysinfo {
    long uptime;              // Seconds since boot
    unsigned long totalram; // Total usable main memory size
    unsigned long freeram;  // Available memory size
    unsigned short procs;   // Number of current processes
};
```

in this file we define our sysinfo struct and it will be included in sysinfo.c so we can upgrade the value of this struct.

2- sysproc.c

```c
uint64
sys_sysinfo(void)
{
  uint64 info; // user pointer to struct stat

  // if( (argaddr_modify(0, &info)) < 0)
  //   return -1;
  argaddr(0, &info);
  return systeminfo(info);
}
```

in this file we define the sysinfo system call which calls sysinfo function.

3- sysinfo.c

xv6-riscv > kernel > C sysinfo.c > systeminfo(uint64)

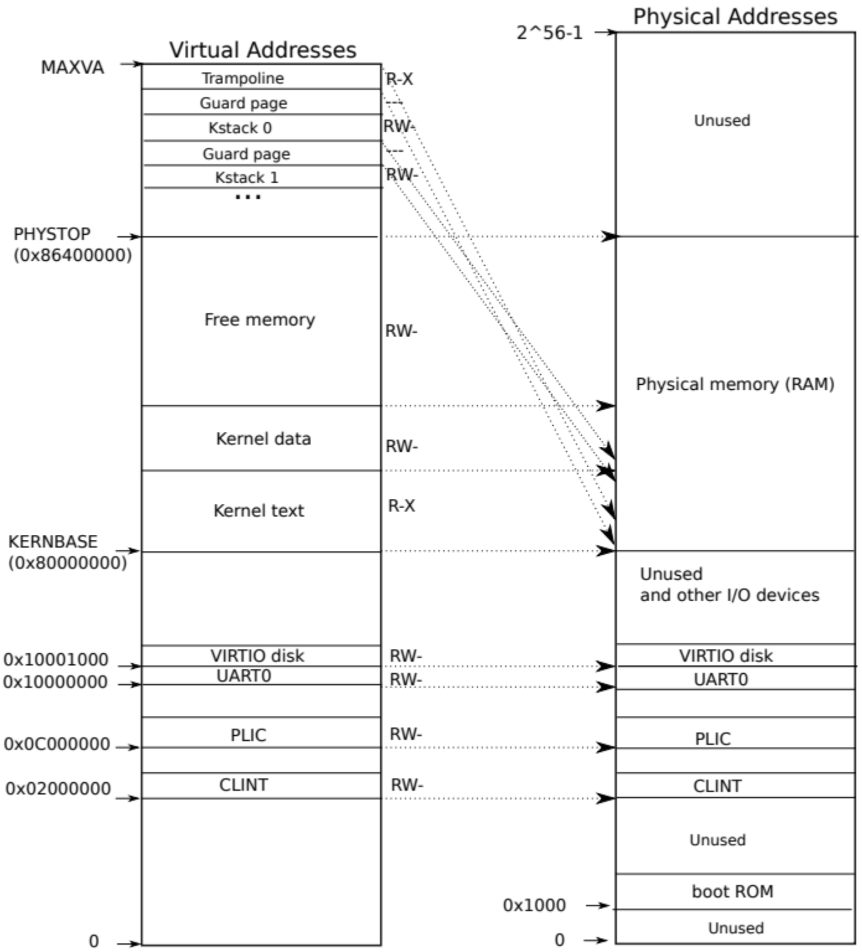
```c
#include "types.h"
#include "riscv.h"
#include "param.h"
#include "memlayout.h"
#include "spinlock.h"
#include "defs.h"
#include "sysinfo.h"
#include "proc.h"

// Get current system info
// addr is a user virtual address, pointing to a struct sysinfo.
int
systeminfo(uint64 addr) {
    struct proc *p = myproc();
    struct sysinfo info;

    info.uptime = ticks;
    info.freeram = freemem();
    info.procs = nproc();
    info.totalram = PHYSTOP - KERNBASE;

    if(copyout(p->pagetable, addr, (char *)&info, sizeof(info)) < 0)
        return -1;
return 0;
}
```

In this file we define the function which is called after system call. This function will update the values of sysinfo struct. First argument is uptime which calculate the ticks that passed since the OS booting and we’ll turn it to second (each tick is approximately 100 msecs). For calculating free memory and total processes we use freemem and nproc function in order (we’ll explain them in future). And for calculating total memory we subtract KERNBASE from PHYSTOP (the reason for that is explained in xv6 tutorial book – see picture below).



4- proc.c

```c
// Get the number of processes whose state is not UNUSED.
int
nproc(void)
{
  int n = 0;
  struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state != UNUSED)
      ++n;
    release(&p->lock);
  }

  return n;
}
```

For calculating processes whose state isn’t UNUSED we define nproc function

In this function we iterate through all processes and if their state is not UNUSED, we will increase n by one then we will return it.

5- kalloc.c

```
// Get the number of bytes of free memory
int
freemem(void)
{
  int n = 0;
  struct run *r;
  acquire(&kmem.lock);

  for (r = kmem.freelist; r; r = r->next)
    ++n;

  release(&kmem.lock);

  return n * 4096;
}
```

In this function in kalloc.c file we iterate through kmem struct and for every freelist in memory we multiply it by 4096 then it will be in bits.

6-defs.h

```
struct sysinfo;
```

```
// kalloc.c
void*           kalloc(void);
void            kfree(void *);
void            kinit(void);
int             freemem(void);

```

```
int             nproc(void);
```

Defining functions in defs,h file.

7-Makefile

```
$K/sysinfo.o \
```

```
$U/_sysinfotest\
```

Changes in Makefile to create sysinfo object file and showing its systemcall

8- sysinfotest.c

xv6-riscv > user > C sysinfotest.c > ...

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

int main(int argc, char **argv){
    struct sysinfo info;
    sysinfo(&info);
    printf("uptime = %d ticks\n", info.uptime);
    // printf("uptime approximately ~= %.1fseconds\n", info.uptime * 0.1);
    printf("Total Ram = %d\n", info.totalram);
    printf("Free Ram = %d\n", info.freeram);
    printf("Procs = %d\n", info.procs);
    printf("successful\n");
    exit(0);
}
```

a test for sysinfo systemcall. You can see the result in picture below:

```
hart 2 starting
hart 1 starting
init: starting sh
$ ls
.              1 1 1024
..             1 1 1024
README         2 2 2354
cat            2 3 32808
echo           2 4 31696
forktest       2 5 15616
grep           2 6 36152
init           2 7 32128
kill           2 8 31688
ln             2 9 31496
ls             2 10 34704
mkdir          2 11 31760
rm             2 12 31744
sh             2 13 53936
stressfs       2 14 32472
usertests      2 15 181768
grind          2 16 47672
wc             2 17 33800
zombie         2 18 31144
sysTest        2 19 31128
ProcTickTest   2 20 31704
sysinfotest    2 21 31688
console        3 22 0
$ sysinfotest
uptime = 128 ticks
Total Ram = 134217728
Free Ram = 133382144
Procs = 3
successful
$ sysinfotest
uptime = 179 ticks
Total Ram = 134217728
Free Ram = 133382144
Procs = 3
successful
$
```