Guide Book for xv6 on ARMv7 (Cortex A)

Zhiyi Huang
Department of Computer Science
University of Otago

February 22, 2018

Contents

Chapter 1

The Boot Procedure

OS kernel like xv6 is a block of code that is loaded and resides in RAM for running once a computer is booted.

xv6 files are compiled, linked, and converted as a binary image (aka. kernel image) that is mapped to the physical memory of RPI2 when loaded. The initial physical address the kernel image is loaded at 0x8000, which has the code at `_start` in *entry.S*.

Therefore, when the xv6 kernel image is loaded into RPI2, the control is transferred by the boot loader to the code at `_start` in *entry.S*.

Now we will go through the code starting from `_start` and get an overview of what xv6 is doing after the control is handed over to it.

The code at `_start` is actually a trap vector which will be explained later. Let us just look at the first branching instruction `b boot_reset` which transfers the control flow to the instruction at `boot_reset`.

At `boot_reset`, there are many ARM assembly instructions, which could put off a lot of newbies who are not familiar with ARM instructions and architectures. However, we don't need to understand precisely the instructions to understand xv6. The instructions are just following some ARM boot pattern to make the hardware ready for executing the C code of xv6. For those who are interested in the gory details, refer to *ARM Cortex-A Series Programmer's Guide*, *ARM Technical Reference Manuals (TRMs)*, and *ARM Architecture Reference Manual (the ARM ARM)*.

Switch to the SVC mode

The following code sets the CPU to SVC mode (Supervisor mode).

```
.set PSR_MODE_SVC, 0x13
.set PSR_MODE_IRQ_DISABLED, (1<<7)
.set PSR_MODE_FIQ_DISABLED, (1<<6)
msr cpsr_c, #(PSR_MODE_SVC + PSR_MODE_FIQ_DISABLED + PSR_MODE_IRQ_DISABLED)
```

ARM has eight modes such as User, FIQ, IRQ, SVC etc. Except that User is unprivileged mode, the other modes are privileged modes. The privileged modes can execute any instructions that configure hardware directly, while the User mode cannot execute those instructions directly but has to use a Supervisor call instruction (SVC) to ask the OS kernel running in the SVC mode to help achieve the equivalent functions.

Therefore, when xv6 is running, it should sit in the SVC mode for user applications running in the User mode to request services with system calls via the Supervisor call instruction (SVC).

Note that RPI2 adopts the ARMv7 architecture, which has the TrustZone Security Extensions. We will not talk about the extensions in detail but should state that xv6 is assuming to run in the Secure world.

Disable caches etc

The following code disables the caches, the Memory Management Unit (MMU), and flow prediction of the running CPU (or core). MMU is the unit that translates virtual addresses into physical addresses in a virtual memory system.

```
mrc p15, 0, r0, c1, c0, 0
bic r0, r0, #(0x1 << 12)  // Disable instruction cache
bic r0, r0, #(0x1 << 11)  // Disable flow prediction
bic r0, r0, #(0x1 <<  2)  // Disable data cache
bic r0, r0, #0x1    // Disable MMU
mcr p15, 0, r0, c1, c0, 0
```

This is a standard practice for booting an OS kernel as we will need to clean the caches to make sure xv6 starts on a clean slate. Note that all cores in RPI2 are running the same code at the moment until the caches are cleaned.

The following code enables the ACTLR.SMP bit.

```
mrc p15, 0, r0, c1, c0, 1
orr r0, r0, #(1 << 6)
mcr p15, 0, r0, c1, c0, 1
```

The ACTLR.SMP bit, once set, indicates this core participates in cache coherency maintenance in the following cache operations.

The following code invalidates TLB and branch prediction caches.

```
mov r0,#0
mcr p15, 0, r0, c8, c7, 0 // Invalidate unified TLB
mcr p15, 0, r0, c7, c5, 6 // Invalidate BPIALL
```

TLB (Translation Lookaside Buffer) is a fast buffer used by the MMU to store the mappings between the virtual address space and the physical address space. Usually such mappings (aka. page tables) are stored in memory (RAM), which is slow for MMU to access. So TLB is needed for better MMU performance. The branch prediction cache is similar to an instruction cache with prefetched instructions.

The following code set the address of the trap vector as `_start`.

```
ldr r0, =_start
mcr p15, 0, r0, c12, c0, 0
```

The trap vector is used to handle interrupts and exceptions of the core. However, the trap vector set at `_start` is only for debugging purposes so we can ignore it for now. We will reset the trap vector for xv6 later.

The following code invalidates the caches. It is not necessary for new Cortex-A processors like RPI2, but we put the code here for compatibility to older Cortex-A processors.

```
// Invalidate l1 instruction cache
mrc p15, 1, r0, c0, c0, 1
tst r0, #0x3
mov r0, #0
mcrne p15, 0, r0, c7, c5, 0

// Invalidate data/unified caches
mrc p15, 1, r0, c0, c0, 1
ands r3, r0, #0x07000000
mov r3, r3, lsr #23
```

```
  beq finished

  mov r10, #0
loop1:
  add r2, r10, r10, lsr #1
  mov r1, r0, lsr r2
  and r1, r1, #7
  cmp r1, #2
  blt skip

  mcr p15, 2, r10, c0, c0, 0
  isb
  mrc p15, 1, r1, c0, c0, 0
  and r2, r1, #7
  add r2, r2, #4
  ldr r4, =0x3ff
  ands r4, r4, r1, lsr #3
  clz r5, r4
  ldr r7, =0x7fff
  ands r7, r7, r1, lsr #13
loop2:
  mov r9, r4

loop3:
  orr r11, r10, r9, lsl r5
  orr r11, r11, r7, lsl r2
  mcr p15, 0, r11, c7, c6,2
  subs r9, r9, #1
  bge loop3
  subs r7, r7, #1
  bge loop2

skip:
  add r10, r10, #2
  cmp r3, r10
  bgt loop1
finished:
```

The following code activates the register TTBR0 for storing the address of the page table for MMU.

```
mov r0,#0x0
mcr p15, 0, r0, c2, c0, 2
```

ARMv7 has two registers TTBR0 and TTBR1 for storing page table addresses. Currently we only use TTBR0 pointing to a single page table including both kernel memory space and user memory space, but it would be nice and efficient to use both TTBR0 and TTBR1, one for the kernel-space page table and one for the user-space page table. Currently we have to copy the user-space page table when switching processes; however, using TTBR1, we can simply make TTBR1 point to the new user-space page table while TTBR0 is always pointing to the kernel-space page table.

The following code sets the attributes of the 16 memory domains supported by ARM.

```
mrc p15, 0, r0, c3, c0, 0
```

```
ldr r0, =0x55555555
mcr p15, 0, r0, c3, c0, 0
```

A domain is similar to the concept of segmentation in x86. The above code sets the 2-bit attribute of each domain to 01 (Clients), which means when any of the domains is accessed, the TLB entries of the domain will be checked to see if the access is permitted by the corresponding page table. This is good as we will use page tables to protect the different memory regions/domains.

The following code sets all cores to wait except the core 0.

```
mrc p15, 0, r0, c0, c0, 5
ands r0, r0, #0x03
wfene
bne mp_continue
```

All cores except core 0 will wait at the WFE instruction until external events happen in those cores. Currently xv6 does not use multiple cores so all cores except core 0 will get stuck at WFE (WFE means Wait for Event and will wait until an external event happens; `wfene` means if the Z conditional flag of the previous instruction is zero then WFE is executed). Since only core 0 does not satisfy the condition (its Z flag equals 1), core 0 will skip WFE as well as the branching instruction (`bne`).

The following code invalidates all page directory (first-level page table) entries (4096 in total) starting at K_PDX_BASE.

```
mmu_phase1:
  ldr r0,=K_PDX_BASE
  ldr r1,=0xfff
  ldr r2,=0
pagetable_invalidate:
  str r2, [r0, r1, lsl#2]
  subs r1, r1, #1
  bpl pagetable_invalidate
```

The following code sets the page directory entries for the initial memory mappings.

```
  ldr r2,=0x14406 // set the entry attributes

  // Map VM address 0x0-0x100000 to physical memory 0-1 M
  ldr r1,=PHYSTART
  lsr r1, #20
  orr r3, r2, r1, lsl#20
  str r3, [r0, r1, lsl#2]

// Map VM address 0x80000000-0x80100000 to PM 0-1 M
ldr r1,=PHYSTART
lsr r1, #20
orr r3, r2, r1, lsl#20
ldr r1,=KERNBASE
lsr r1, #20
str r3, [r0, r1, lsl#2]

// Map device memory (just GPIO for LED debug)
ldr r2,=0xc16 //attributes for device memory
ldr r1,=(MMIO_PA+0x200000)
lsr r1, #20
```

```
orr r3, r2, r1, lsl#20
ldr r1,=(MMIO_VA+0x200000)
lsr r1, #20
str r3, [r0, r1, lsl#2]
```

Each entry in the page directory table sets attributes for a mapping of memory region with 1 MB. The mapping allows more partitions of the 1 MB with a second level page table (with a page size 4 KB covered by each entry in the second level page table). However, the mapping also allows large page size like 1 MB or 16 MB. With the entry attributes 0x14406 used above, the binary code is 00010-100-01-0-0000-0-01-10, which matches the attribute pattern Z0GSA-TEX-AP-P-DOMN-X-CB-10. The attributes mean that a large page size 1 MB is used, that the page belongs to domain 0, that the page is bufferable but not cacheable, that the page is readable/writable but for privileged access only, that the page is normal memory type but not cacheable in L2 cache, and that the page is executable, global, etc. You may feel a bit confused but it is normal if you do not quite understand the attributes as they are architecture specific. The key point is that we should set the right attributes for each page (entry) to protect the page and to make the MMU function correctly on the page.

In the code above, both VM regions (0x0-0x100000 and 0x80000000-0x80100000) are mapped to the same physical memory region (the first MB). The reason is as follows. MMU is not enabled yet at the moment so the assembly code e.g. program counter is assuming physical addresses are used. So the first mapping (0x0-0x100000 to 0x0-0x100000) allows the assembly code to work correctly without violating its assumption when MMU is enabled and the page table takes effect as MMU will translate the addresses into the same (equivalent) physical addresses according the first mapping. However, the C code is compiled and linked into the VM region above 0x80000000. To make the C code work properly, we need the second mapping (0x80000000-0x80100000 to 0x0-0x100000) to map the VM addresses assumed by the C code into the physical addresses it resides. Of course, at the moment, we only maps the first MB of the kernel image which is sufficient for running the very first part of the C code. You will see shortly the following C code will extend the mapping of VM region (above 0x80000000) to the whole physical memory space. Also the first mapping will be torn down after the C code takes over as it is no more needed by the rest of the kernel code.

The above third mapping is for device memory. The attributes are set as 0xc16, which is for device memory. You can find the details from *Chapter 9 The Memory Management Unit of ARM Cortex-A Series Programmer's Guide*. We only mapped the device memory for the GPIO device of RPI for debugging purpose. The whole device memory region will be mapped shortly in the C code.

Now that we have done the basic page table, the following code enables MMU etc and makes the C code ready to run.

```
ldr sp, =(KERNBASE+0x3000)
dsb
ldr r1,=_pagingstart
mrc p15, 0, r0, c1, c0, 0
orr r0, r0, #(0x1 << 13)  // High address range for trap vector
//orr r0, r0, #(0x1 << 12)  // Enable I$
//orr r0, r0, #(0x1 << 11)  // Enable flow prediction
//orr r0, r0, #(0x1 <<  2)  // Enable D$
orr r0, r0, #0x1    // Enable MMU
mcr p15, 0, r0, c1, c0, 0
bx r1
  .section .text
.global _pagingstart
_pagingstart:
  bl cmain  // call C functions now; no return if normal
```

```
bl NotOkLoop
```

To run C code, apart from the VM space, we need a stack space for executing C functions. So we set the stack pointer (register) `sp` to 0x80003000 (KERNBASE+0x3000). The space under 0x80003000 (equivalent to physical address space 0x0-0x3000) is available and will be used by the kernel initialization code and the process scheduler code as a stack space.

The above code also chooses the high address range (0xFFFF0000 to 0xFFFF001C) for storing the trap vector. ARMv7 allows high or low address range for storing the trap vector. The low address range is 0x00000000 to 0x0000001C. According to the setting of the 13th bit in the cp15 c1 control register, the core will find the trap/interrupt handlers from the low or high address range. xv6 chooses to put the trap vector into the high range as the low VM address space will be used by user processes. Just in case you are curious, all hardware configurations are completed via a control coprocessor 15 (CP15) in ARM architecture. CP15 configures caches, MMU, etc and provides status information like system performance. CP15 is different for different ARM architectures. Refer to ARM ARM for more details.

Note that we do not enable instruction and data caches yet due to some complication in process context switching. This is a job to do in the near future.

Finally, after MMU is enabled, we jump to the address `_pagingstart`. Since we cannot directly change the program counter (register r15), we used `bx r1` to jump to `_pagingstart` and changed the program counter to the address of `_pagingstart`, which is compiled and linked into the VM space starting at 0x80000000. For details of the VM space range different code sections are linked into, refer to the kernel linker script *kernel.ld*. To understand the linker script, you may need to read *Emprog ThunderBench Linker Script Guide* or any other similar guides.

Now we get into the C world with `bl cmain` and work in the VM space above 0x80000000.

The `cmain()` function is in the file *main.c*.

In `cmain()`, we further configures MMU with the following two functions.

```
mmuinit0();
machinit();
uartinit();
dsb_barrier();
consoleinit();
cprintf("\nHello␣World␣from␣xv6\n");
kinit1(end, P2V((8*1024*1024)+PHYSTART));
// collect some free space (8 MB) for imminent use
// the physical space below 0x8000 is reserved for PGDIR and kernel stack
kpgdir=p2v(K_PDX_BASE);
mailboxinit();
pm_size = getpmsize();
mmuinit1();
```

In the xv6 code, memory mapping (via configuring MMU) is achieved in two stages: `mmuinit0()` and `mmuinit1()`. The reason is that the memory size is usually unknown in a system so only a small portion (a few MBs) of the physical memory is mapped. Then after verb|mmuinit0()|, since the device memory is mapped, we can find out the real size of the physical memory through querying the peripherals. So in `mmuinit1()` we can map the whole physical memory into the VM space.

In `mmuinit0()` (in file *mmu.c*), the following code maps the known minimum physical memory PHYSIZE (256 MB) to the kernel VM space (the address above 0x80000000).

```
va = KERNBASE + MBYTE;
for(pa = PHYSTART + MBYTE; pa < PHYSTART+PHYSIZE; pa += MBYTE){
  l1[PDX(va)] = pa|DOMAIN0|PDX_AP(K_RW)|SECTION|CACHED|BUFFERED;
  va += MBYTE;
```

```
}
```

`l1` is the base address of the page directory table used before. Now we are filling more entries. Each entry has the page attributes which contain the 10 most significant bits of the physical address (`pa`), the domain (DOMAIN0) the page belongs, kernel readable/writable, large page size 1 MB (SECTION), cacheable and bufferable. Note that if the caches are not enabled, the cacheable attribute has no effect.

With this mapping, we can access all the physical memory (256 MB) in the kernel VM space (0x80000000 to 0x80000000+256MB). Since the kernel image only occupies a small portion of the memory space, the rest of the memory will be organized as free space for future kernel data structures and user code/data structures.

The following code in `mmuinit0()` maps the I/O (or device) memory.

```
va = MMIO_VA;
for(pa = MMIO_PA; pa < MMIO_PA+MMIO_SIZE; pa += MBYTE){
  l1[PDX(va)] = pa|DOMAIN0|PDX_AP(K_RW)|SECTION;
  va += MBYTE;
}
```

The I/O memory on RPI2/3 has 16 MB (MMIO_SIZE) starting at physical address MMIO_PA (0x3F000000). It is mapped to the kernel VM space starting at MMIO_VA=0xD0000000. Since it is I/O memory, it is not cacheable and bufferable. The reason is that I/O memory has the side-effect of controlling devices, and if it is buffered or cached, the expected side-effect will not happen.

Then we map the GPU memory of RPI2, which is also a device memory.

```
va = GPUMEMBASE;
for(pa = GPUMEMBASE; pa < (uint)GPUMEMBASE+(uint)GPUMEMSIZE; pa += MBYTE){
  l1[PDX(va)] = pa|DOMAIN0|PDX_AP(K_RW)|SECTION;
  va += MBYTE;
}
```

We used the GPU memory for displaying the console messages on the monitor connected by the HDMI cable. However, since the GPU has changed on RPI2 and the specification of the hardware is not open, we cannot make the GPU framebuffer working. Therefore, the above code has no use and the console message can only be displayed via the serial port (the `uart` port).

Finally in `mmuinit0()`, the VM space at HVECTORS (0xFFFF0000) for the trap vector is mapped to physical address 0x0 in the following code.

```
va = HVECTORS;
l1[PDX(va)] = (uint)l2|DOMAIN0|COARSE;
l2[PTX(va)] = PHYSTART|PTX_AP(K_RW)|SMALL;
```

A second-level page table (`l2`) is used in this case. A COARSE second-level page table has 256 entries for a page size 4 KB. The MMU also allows a finer page size 1 KB but xv6 chooses to use a page size of 4 KB. Similarly, the attribute SMALL in the second-level page table means the page size is 4 KB.

You may still remember we have set the high address range (at HVECTORS=0xFFFF0000) for the trap vector in *entry.S*. In `tvinit()` in *trap.c*, we will copy the xv6 trap vector to HVECTORS before enabling interrupts.

Back to the following code in `cmain()`, we can see other initialization functions.

```
mmuinit0();
machinit();
uartinit();
dsb_barrier();
consoleinit();
```

```
cprintf("\nHello␣World␣from␣xv6\n");
kinit1(end, P2V((8*1024*1024)+PHYSTART));
// collect some free space (8 MB) for imminent use
// the physical space below 0x8000 is reserved for PGDIR and kernel stack
kpgdir=p2v(K_PDX_BASE);
mailboxinit();
pm_size = getpmsize();
mmuinit1();
```

`machinit()` initializes the data structures for multiple cores/CPUs. The current xv6 port only uses one CPU so only the first element of the array will be used.

`uartinit()` initializes the serial port (mini UART) to be used by the console. The function is defined in *uart.c* as below.

```
void uartinit(void)
{
        outw(AUX_ENABLES, 1);
        outw(AUX_MU_CNTL_REG, 0);
        outw(AUX_MU_LCR_REG, 0x3);
        outw(AUX_MU_MCR_REG, 0);
        outw(AUX_MU_IER_REG, 0x1);
        outw(AUX_MU_IIR_REG, 0xC7);
        outw(AUX_MU_BAUD_REG, 270); // (250,000,000/(115200*8))-1 = 270

        setgpiofunc(14, 2); // gpio 14, alt 5
        setgpiofunc(15, 2); // gpio 15, alt 5

        outw(GPPUD, 0);
        delay(10);
        outw(GPPUDCLK0, (1 << 14) | (1 << 15) );
        delay(10);
        outw(GPPUDCLK0, 0);

        outw(AUX_MU_CNTL_REG, 3);
        enableirqminiuart();
}
```

This code shows how a device driver would look like. With `inw()` and `outw()`, it just reads/writes the device memory (registers) to interact with the device and configure the device. The above code configures the features of the serial port such as the speed of the port (baudrate=115200), configures the GPIO pins 14 and 15 as UART pins, and then enables the interrupt of the UART port. More details about UART will be given in Chapter 5. Though you do not need to understand the details of `uartinit()` to understand xv6, you are encouraged to understand the details by reading *Chapter 2 Auxiliaries: UART1 & SPI1, SPI2* and *Chapter 6 General Purpose I/O (GPIO)* of *BCM2835 ARM Peripherals*. Also the UART device driver is very basic. You are welcome to make it full-fledged.

Back to the code of verb | cmain() | :

```
mmuinit0();
machinit();
uartinit();
dsb_barrier();
consoleinit();
```

```
cprintf("\nHello World from xv6\n");
kinit1(end, P2V((8*1024*1024)+PHYSTART));
// collect some free space (8 MB) for imminent use
// the physical space below 0x8000 is reserved for PGDIR and kernel stack
kpgdir=p2v(K_PDX_BASE);
mailboxinit();
pm_size = getpmsize();
mmuinit1();
```

`dsb\_barrier()` is a memory barrier to flush the content in data caches into the memory and to guarantee all previous memory accesses are completed. Since the caches are not enabled, this memory barrier is not necessary for the current port, but we leave it here for future extensions.

`consoleinit()` initializes the xv6 console. A console is the place where kernel messages are displayed. Also it is the place for the command shell to interact with the user in xv6.

```
void consoleinit(void)
{
  uint fbinfoaddr;
  fbinfoaddr = initframebuf(framewidth, frameheight, framecolors);
  if(fbinfoaddr != 0) NotOkLoop();

  initlock(&cons.lock, "console");
  memset(&input, 0, sizeof(input));
  initlock(&input.lock, "input");

  memset(devsw, 0, sizeof(struct devsw)*NDEV);
  devsw[CONSOLE].write = consolewrite;
  devsw[CONSOLE].read = consoleread;
  cons.locking = 1;
  panicked = 0; // must initialize in code since the compiler does not

  cursor_x=cursor_y=0;
}
```

In the above code, we first initialize the framebuffer of the GPU. Since the framebuffer is not working, we can ignore it for now. Then the data structures `cons` and `input` are initialized, especially the locks. The locks are used to prevent data races between the scheduler and the interrupt handlers. More details will be explained when the locks are used. `input` is used to contain input characters from the console (i.e. UART). It uses a circular buffer with a few indices. More details of how the console works will be explained when *console.c* is studied. Next, the device structure array `devsw` is initialized. Currently there is only one device (CONSOLE==1) used in the array. It could be extended for other devices like an IDE disk. The console read/write functions `consolewrite()` and `consoleread()` are registered to the device structure and will be called by the kernel when the console is needed. Other variables are initialized but we can ignore them for now.

Back to `cmain()`:

```
mmuinit0();
machinit();
uartinit();
dsb_barrier();
consoleinit();
cprintf("\nHello World from xv6\n");
```

```
kinit1(end, P2V((8*1024*1024)+PHYSTART));
// collect some free space (8 MB) for imminent use
// the physical space below 0x8000 is reserved for PGDIR and kernel stack
kpgdir=p2v(K_PDX_BASE);
mailboxinit();
pm_size = getpmsize();
mmuinit1();
```

Since the console is ready, we can use `cprintf()` to print messages to the console from now on.

Then we use `kinit1()` to collect the free space within the first 8 MB physical memory space. The first parameter of `kinit1()` is the start address of the free space and the second parameter is the end address. In `kinit1(end, P2V((8*1024*1024)+PHYSTART))`, `end` is the address where the kernel image ends in the VM space. So basically `kinit1(end, P2V((8*1024*1024)+PHYSTART))` will collect the free space after the kernel image until the first 8 MB of the kernel mapped space. Note that there are some free space below 0x8000 (where the kernel image starts) which is not put into the free page list. Those pages in the free space are used by the trap vector, the kernel stack, and the page tables.

```
void kinit1(void *vstart, void *vend)
{
  initlock(&kmem.lock, "kmem");
  kmem.use_lock = 0;
  kmem.freelist = 0;
  freerange(vstart, vend);
}
```

The above `kinit1()` initializes the `kmem` structure including the lock and `freelist`. The use of `use_lock` is a bit tricky but I leave it for you to figure out later.

`freerange(vstart, vend)`, as below, puts the pages between `vstart` and `vend` into `freelist` by calling `kfree()`. I leave it for you to read `kfree()` as it is just a standard list management function for free pages.

```
void freerange(void *vstart, void *vend)
{
  char *p;
  p = (char*)PGROUNDUP((uint)vstart);
  for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
    kfree(p);
}
```

Then the following code in `cmain()` sets `kpgdir` to the address of the page directory table (the first-level page table).

```
kpgdir=p2v(K_PDX_BASE);
mailboxinit();
pm_size = getpmsize();
mmuinit1();
```

Then we initialize the mailbox between the CPU and the GPU in RPI2. In Raspberry Pi, the system is booted by the GPU and then handed over to the CPU. So the GPU has the system information such as the memory space allocated to the CPU etc. To collect the information, the CPU has to use a mechanism called mailbox to communicate with the GPU. Since mailbox in Raspberry Pi is proprietary and not fully open for developers, we will not dive into the details. For those who are interested, refer to *https://github.com/raspberrypi/firmware/wiki/Mailboxes*.

One key use of mailbox in xv6 is to find out the available memory space allocated to the CPU. In `getpmsize()`, we get the physical memory size using the mailbox. The returned physical memory size is stored in `pm_size`, which will replace PHYSTOP and PHYSIZE wherever needed.

Then we can map the rest of the physical memory (above 256 MB) into the kernel VM space in `mmuinit1()`.

```
void mmuinit1(void)
{
  pde_t *l1;
  uint va1, va2;
  uint pa, va;

  l1 = (pde_t*)(K_PDX_BASE);

  // map the rest of RAM after PHYSTART+PHYSIZE
        va = KERNBASE + PHYSIZE;
        for(pa = PHYSTART + PHYSIZE; pa < PHYSTART+pm_size; pa += MBYTE){
                l1[PDX(va)] = pa|DOMAIN0|PDX_AP(K_RW)|SECTION|CACHED|BUFFERED;
                va += MBYTE;
        }

  // undo identity map of first MB of ram
  l1[PDX(PHYSTART)] = 0;

  // drain write buffer; writeback data cache range [va, va+n]
  va1 = (uint)&l1[PDX(PHYSTART)];
  va2 = va1 + sizeof(pde_t);
  va1 = va1 & ~((uint)CACHELINESIZE-1);
  va2 = va2 & ~((uint)CACHELINESIZE-1);
  flush_dcache(va1, va2);

  // invalidate TLB; DSB barrier used
  flush_tlb();
}
```

After the available physical space is mapped, we tear down the identity mapping of the first MB as it is only needed by the initial assembly code and also the first MB of the VM space will be used by the user process address space in xv6.

Finally, `mmuinit1()` flush the data cache and the TLB as the identity mapping is invalid and the corresponding cache entries and the TLB entries should be invalidated.

Now `cmain()` is executing a list of initialization functions.

```
  pinit();
  tvinit();
  binit();
  fileinit();
  iinit();
  ideinit();
  kinit2(P2V((8*1024*1024)+PHYSTART), P2V(pm_size));
  userinit();
  timer3init();
```

```
scheduler();
```

`pinit()` in *proc.c* initializes the process table `ptable` (an array of process control block) and its lock. Since there is no process yet, the table is set empty. We will look at the detail of a process control block in Chapter 2.

`tvinit()` in *trap.c* copies the xv6 trap vector to the address HVECTORS (0xFFFF0000) so that when interrupts and exceptions occur the CPU can find the right handlers provided by xv6. Likewise, the caches have to be flushed to the RAM as the interrupt mechanism will get the handlers directly from the RAM instead of the caches. Finally `tvinit()` sets up the stack space for the other ARM processor modes like IRQ and FIQ. When an interrupt happens, the processor switches to the IRQ or FIQ mode depending on if it is a normal IRQ or a fast IRQ. So we should prepare a stack space for each such mode. However, since user processes use system calls (via the SVC instruction) enters the SVC mode, we will face the complication of handling multiple modes in our interrupt/trap/exception handlers if we want to handle them with the same handlers. We will discuss the complication when we look at the trap handlers in Chapter 4.

`binit()` in *bio.c* initializes the buffer cache `bcache` for disk blocks. `bcache` is a doubly linked list of buffer structures holding copies of disk blocks. Using the buffer cache in RAM reduces the number of disk accesses and improves the performance of disk-based file systems. We will look at the buffer cache in detail in Chapter 7.

`fileinit()` in *file.c* initializes the file table `ftable`. `ftable` contains the information (e.g. inode) of all the files opened by the user processes. An open file is defined by `struct file` in *file.h*. More details of the file operations will be found in Chapter 7.

`iinit()` in *fs.c* initializes the inode cache `icache`. `icache` stores copies of inodes in RAM to improve the performance of file operations in disk-based file systems. When a file is retrieved, its inode (indexing node) is frequently accessed. Usually inodes reside on the disk as do the data blocks of the files. Putting the inode in RAM once a file is opened can greatly accelerate the data retrieval in file operations. More details will be found in Chapter 7.

`ideinit()` in *memide.c* sets the parameters of the memory disk used in xv6. Since the memory disk is built into the kernel image, we can easily find its start address (`memdisk`) and size (`disksize`). In the code below, `_binary_fs_img_start` and `_binary_fs_img_end` are global variables defined in *entry.S* and set by the linker at linking time.

```
void ideinit(void)
{
  memdisk = _binary_fs_img_start;
  disksize = div(((uint)_binary_fs_img_end - (uint)_binary_fs_img_start), 512);
}
```

`kinit2(P2V((8*1024*1024)+PHYSTART), P2V(pm_size))` puts into the freepage list the pages after the 8 MB until the end of the available memory space.

`userinit()` in *proc.c* creates the first user process in the process table and makes it ready to run. This process is very simple though it is manually set. It will be the ancestor of all processes in the system. The code of this process is in *uprogs/initcode.S* and shown as below.

```
.globl start
start:
  push {lr}
  ldr r0, =argv
  push {r0}
  ldr r0, =init
  push {r0}
  mov r0, #SYS_exec
```

```
swi #T_SYSCALL
pop {lr}
pop {lr}
pop {lr}
bx lr
```

Though the assembly code looks intimidating, the code basically calls a system call `exec()` with the parameter `init`. That means the first process just loads another program called `init`. `init` (compiled from *uprogs/init.c*) will create the user-space console device and fork a child process executing the command shell. With the command shell running with the console, the user can interact with the system using the commands of the system. More details of the first process will be described in Chapter 2.

Note that the above first process is not running yet. It is only set up and made ready to run by `userinit()`. It has to wait a little while for the process scheduler `scheduler()` to take effect.

`timer3init()` in *timer.c* enables the timer interrupt for the system timer device of RPI2. It also sets the timer interrupt frequency to 100 time per second. This timer interrupt handling is very important for the process scheduler as it allows the scheduler to take a running process off from the CPU if it hogs the CPU for 10 ms. Note that all interrupts are blocked until the following scheduler function starts.

`scheduler()` in *proc.c* runs an endless for loop. It searches any runnable process to run. Since the first process is in the process table, the scheduler selects the process and switches to the context of the process set by `userinit()`. The context of the scheduler is saved and will be restored when the process calls scheduling related functions such as `sched()`, `sleep()`, and `yield()` in the kernel or it is deprived of the CPU by the timer interrupt handler.

Now you can get an idea of the dynamic activities in xv6. At the foreground, the scheduler is running endlessly selecting a process to run and then switch to the process context. After some execution time of the process, the control is given back to the scheduler by the scheduling related functions switching to the scheduler context.

At the background, the interrupt handlers will be executed now and then to interrupt the scheduler or the process. But they will restore the context of the interrupted scheduler or process after the interrupts are handled.

Note that the key to understand how context is switched is that every process has its own stack space when running in kernel space (in addition to its user space stack) and the scheduler has its own stack space which is the same stack as the booted SVC mode. Basically the stack space of the SVC mode is switched between the scheduler's stack and the processes' stacks. The interrupt handlers use one of those stacks (either the scheduler's stack or the kernel stack of the current running process) to store the interrupted context, i.e., the registers.

We will explain how the above activities interact with each other in more detail in the following chapters.

Chapter 2

The First Process

A process is a running program managed by the kernel. It provides the user an illusion of abstract machine that the process exclusively owns. Each process has its own CPU state (e.g., registers) and memory space that other processes cannot access, so that it appears the process owns the whole computer system to execute its program.

xv6 manages a process using a data structure called `proc` as below. It is usually called Process Control Block (PCB).

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  volatile int pid;            // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

`sz` is the size of the process memory space which can be extended or reduced by a system call `sbrk()`. The process memory space consists of a code section, a section of global variables, a stack, and a heap which can grow or shrink by `sbrk()`. The process space starts at 0 and can maximumly reach 0x80000000, as you may recall the space above 0x80000000 belongs to the kernel space. We will look at the process memory space in more detail when we explain the system call `exec()` in Chapter 3.

`pgdir` is a pointer to the process page table. Each process has its own page table to define its memory space. However, the page table entries for the space above 0x80000000 are the same for all processes and define the kernel space which the processes cannot access. Using the same page table for both the kernel space and the process memory space makes the kernel code easily access the process memory space without modifying the page table.

`kstack` is the pointer to the kernel stack of the process. Each process has two stacks: one for the user space which is allocated by `exec()`, and the other is the kernel stack allocated by `fork()` when a process is created. The kernel stack is used by the kernel code when the process calls a system call or interrupted

by interrupt handlers. It is the place to store and restore the execution context of the process when a system call or interrupt occurs. More details will be explained in Chapter 4.

`state` is the process state which could be SLEEPING, RUNNABLE, RUNNING, etc. The process scheduler uses it to manage the different states of a process. For example, the current process on CPU is in the RUNNING state, the processes that are ready to run have the RUNNABLE state, and the processes that are waiting for events are in the SLEEPING state. There are other states for special cases which will be explained in Chapter 6.

`pid` is the process ID which is a unique number. The number is keep increasing so the maximum number of the `pid` is 2,147,483,647 which is sufficient for a small system.

`parent` is pointing to the process that created the process with `fork()`. Every process has a parent process except the first process which we will discuss about shortly.

`tf` is pointing to the trap frame for a system call or interrupt. A trap frame contains the contents of the registers when a system call or interrupt occurs. It is on the above kernel stack of the process. More details will be explained in Chapter 4.

`context` is the kernel execution context where the process is switched off from the CPU. When the process is to execute again, the context will be restored. Similar to the trap frame, it has the contents of registers. The difference between the trap frame and the context is that the trap frame includes more registers than the context. The context switching happens in the kernel space (the SVC mode), while the trap frame involves a mode switching, e.g., from the USER mode to the SVC mode in ARM architectures. More details will be explained in Chapter 4.

`chan` is a variable used to suspend and wake up the process if it is not zero. It is pointing to a memory address the process is sleeping on and will be waken up at the same address. It is the meeting place for the sleeper and the waker. More details will be explained in Chapter 6.

`killed` means the process is killed if it is non-zero. When the scheduler chooses a process to run, it should check this flag just in case the process is killed by other processes or the user.

`ofile[]` is an array of opened files. The information of opened files are kept in memory to increase performance. More details will be explained in Chapter 7.

`cwd` is pointing to the index node of the current working directory. Again it is for the convenience and performance of the file system. More details will be explained in Chapter 7.

Finally, `name` is the name of the process. It is just for debugging purposes.

In modern OS like Linux, there are dozens of variables for a process structure. However, xv6 has the most essential variables for a process of a small OS.

As we know previously the first process is created by `userinit()` as below before the scheduler is running.

```
_binary_initcode_size = (uint)_binary_initcode_end - (uint)_binary_initcode_start;
p = allocproc();
initproc = p;
if((p->pgdir = setupkvm()) == 0)
  panic("userinit: out of memory?");
inituvm(p->pgdir, _binary_initcode_start, _binary_initcode_size);
p->sz = PGSIZE;
memset(p->tf, 0, sizeof(*p->tf));
p->tf->spsr = 0x10;
p->tf->sp = PGSIZE;
p->tf->pc = 0;  // beginning of initcode.S
safestrcpy(p->name, "initcode", sizeof(p->name));
p->cwd = namei("/");
p->state = RUNNABLE;
```

From the code above, we can see the major operations for creating the first process are in the functions `allocproc()`, `setupkvm()` and `inituvm()`. The rest of the code is just to set the memory size, trap frame, current working directory, and the state of the process. The *i*nitcode.S is set to begin at the address 0 in the process address space by `inituvm()`. The process memory size is just one page (4096 bytes) as *i*nitcode.S is very small and only designed to invoke an `exec()` system call, as shown in the previous chapter.

The trap frame of the process is mostly set to 0 by `memset()` except a few fields like `spsr`, `sp` and `pc`. The process is created by `allocproc()` as if a `fork()` system call is invoked and a child process is created. Now the first process is ready to returned from the `fork()` system call to execute the code of the child process in the user space.

The fields `spsr`, `sp` and `pc` of the trap frame are set as below.

```
p->tf->spsr = 0x10;
p->tf->sp = PGSIZE;
p->tf->pc = 0;  // beginning of initcode.S
```

The above settings make sure that the process will be back to the user space (decided by `spsr = 0x10`), and execute the code at address 0 (which is the beginning of `initcode.S`) with the user stack starting at the top end of the one-page process (at address 4096). More detail of the trap frame will be explained in Chapter 4.

Now we will look at the detail of `allocproc()`. It first finds an unused slot in the PCB table `ptable`. The lock functions `acquire()` and `release()` will disable and enable interrupts in order to avoid data race and deadlock. More detail of locks will be explained in Chapter 8.

```
acquire(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  if(p->state == UNUSED)
    goto found;
release(&ptable.lock);
return 0;
```

After an empty slot is found, the state and process ID of the process are set as below.

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  release(&ptable.lock);
```

The state of EMBRYO just means the slot is not available and the process is under construction. The process ID is just set to the next available ID number in sequence. Then the lock is released to allow other activities like interrupt handlers to access the PCB table.

The following code allocates the kernel stack for the process. `kalloc()` returns one page available at the beginning of the free page list (refer to *k*alloc.c for detail). If there is no free page available, the PCB slot is set as unused and the process creation fails.

```
if((p->kstack = kalloc()) == 0){
  p->state = UNUSED;
  return 0;
}
memset(p->kstack, 0, PGSIZE);
sp = p->kstack + KSTACKSIZE;
```

After the kernel stack is successfully allocated, its space is set to 0 so that any junk data left previously in the allocated page will have no effect on the use of the stack. Then the stack pointer `sp` is set to the top end of the page and the trap frame and the process context are arranged on the stack as below.

```
sp -= sizeof *p->tf;
p->tf = (struct trapframe*)sp;

sp -= sizeof *p->context;
p->context = (struct context*)sp;
memset(p->context, 0, sizeof *p->context);
p->context->pc = (uint)forkret;
p->context->lr = (uint)trapret;
return p;
```

The trap frame is arranged first on the stack and `tf` of the process is made to point to the space of the trap frame. Note that data structures are arranged from the low address location to the high address location, though the stack pointer moves reversely.

Then the process context is arranged on the stack and `context` of the process is made to point to the right location on the stack. The space of the context is set to 0 except `pc` and `lr`. `pc` of the context is the instruction to be executed once the scheduler switch to this process. It is set to the function `forkret()`. `lr` of the context is the location of the execution when the current function (i.e. `forkret()`) returns. It is now set to `trapret()`. In summary, when the process is scheduled to run, it will execute `forkret()` and then execute `trapret()`. These two functions basically lead the process to execute the user space code, i.e. *initcode.S* for the first process, once the scheduler switches the process to the CPU.

You may find the following code of the `forkret()` function very simple.

```
static int first = 1;
release(&ptable.lock);
if (first) {
  first = 0;
  initlog();
}
```

Since the scheduler is holding the lock for the PCB table when doing context switching, the first thing the switched process should do is to release the lock. If this is the first process ever, some initialization functions that are likely to sleep like `initlog()` are executed here. `initlog()` initialises the logging system of the file system. The logging system is to guarantee the atomicity of the file operations and consistency of the file system. More detail of the logging system will be shown in Chapter 7.

The `trapret()` function in *e*xception.S is written in assembly. It simply restores the registers of the trap frame set previously by `userinit()` and returns to the user space. In the case of the first process, *i*nitcode.S will be executed from the beginning.

The data structures for the trap frame and context are architecture dependent as they contain the CPU registers. For RPI2, below are the definitions of the data structures.

```
struct trapframe {
  uint sp; // user mode stack pointer
  uint r0;
  uint r1;
  uint r2;
  uint r3;
  uint r4;
  uint r5;
  uint r6;
  uint r7;
  uint r8;
  uint r9;
```

```
  uint r10;
  uint r11;
  uint r12;
  uint r13;
  uint r14;
  uint trapno;
  uint ifar; // Instruction Fault Address Register (IFAR)
  uint cpsr;
  uint spsr; // saved cpsr from the trapped/interrupted mode
  uint pc; // return address of the interrupted code
}

struct context {
  uint r4;
  uint r5;
  uint r6;
  uint r7;
  uint r8;
  uint r9;
  uint r10;
  uint r11;
  uint r12;
  uint lr;
  uint pc;
};
```

Compared with the context structure, the trap frame has more registers including CPU mode registers like `cpsr` and `spsr` as system calls and interrupts involve mode switching (e.g. USER to SVC or vice verse). But for context switching, it happens in the SVC mode and involves no mode switching so only the callee-saved registers are stored or restored and the stack space is switched between the scheduler and the process in context switching.

For more detail of context switching, refer to Chapter 6. If you don't quite understand why everything works if `pc` and `lr` are set to `forkret()` and `forkret()` respectively, you need to read materials on ARM function calling conventions. Here is a link of such materials.

Note that the `fork()` system call uses the same `allocproc()` to create a new process. Actually `userinit()` is very similar to the `fork()` function except `userinit()` creates the very simple first process so it omits some steps like duplication of opened files. However, since it creates the first process, `userinit()` has to create the page table and memory space from scratch, while `fork()` only needs to copy the page table and memory space from the parent process. Other than these differences, `userinit()` and `fork()` are doing similar things.

From the above description, you may have a better understanding of how the first process is set ready to run in terms of context switching and code execution. In the following chapter, we will look at how the page table and the memory space are established by `setupkvm()` and `inituvm()` which are defined in *v*m.c.

Chapter 3

Process Memory Space

As mentioned before, the 4 GB Virtual Memory (VM) space is divided into two parts. The higher part (above 0x80000000) is for the kernel space, and the lower part (0x0 - 0x80000000) is for the process address space. The page tables of all processes share the same entries of the kernel space in the first-level page table, though they cannot access the kernel space.

In ARM architectures, the first-level page table consists of four pages. The first two pages have the entries for the process address space and the last two pages are for the kernel space. Each entry has four bytes describing the attributes of a 1 MB memory space. For example, the first entry in the first-level page table describes the attributes of the address range 0x0-0xFFFFF, and the second entry describes the address range 0x100000-0x1FFFFF, and so on. If larger page sizes are used like 1 MB and 16 MB, the entries of the first-level page table directly contain the base address of the mapped physical address in addition to the attributes we described in Chapter 1. If the smaller page sizes like 4 KB and 1 KB are used, the entries point to a second-level page table which contains 256 or 4096 entries. We will go through the functions of establishing first-level and second-level page table entries shortly.

In the xv6 RPI2 port, since the process address space of every program is usually very small, we only use for each process one page for the first-level page table (also called page directory table) which covers the VM address range 0-1 GB. In process context switching, we copy this one-page table to the first page of the kernel page directory table starting at the physical address K_PDX_BASE (0x4000). As mentioned in Chapter 1, ARMv7 has two registers TTBR0 and TTBR1 for pointing to the page directory table. It would be better to use both TTBR0 and TTBR1, one for the kernel-space part of the page directory table and one for the user-space part of the page directory table, so that we can simply make TTBR1 point to the new user-space page table without copying it in context switching, while TTBR0 is always pointing to the kernel-space page table.

With the above understanding in mind, now let us have a look at `setupkvm()` and `inituvm()`, which establish the process address space by creating the user-space page tables.

```
pde_t* setupkvm(void)
{
  pde_t *pgdir;
  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  return pgdir;
}
```

`setupkvm()` is a simple function as shown above. In some other architectures like x86, it involves copying of the kernel-space page directory entries to the process local page table. We are using the opposite strategy: copying the user-space (i.e. process space) entries to the kernel page directory table in

context switching in order to save memory pages for process page tables. This is because, if we keep a copy of the kernel-space entries in each process, we will need four pages for the page directory table for each process. With the opposite strategy, we only need one page for each process.

Therefore, in `setupkvm()`, one page is allocated for the page directory entries of the process address space. Note the page is set to 0 and will be given to `inituvm()` which will set up the entries properly.

For the first process, `userinit()` calls `inituvm()` as below:

```
inituvm(p->pgdir, _binary_initcode_start, _binary_initcode_size);
```

`inituvm()` will establish the entries with the page directory table pointed by `p->pgdir` for the address space starting at `_binary_initcode_start` with a range size `_binary_initcode_size`. Basically `inituvm()` will create proper entries for the address space of the first process and copy `initcode` (compiled from *i*nitcode.S) to the right place of the space for execution. The detail of `inituvm()` is as below.

```
void inituvm(pde_t *pgdir, char *init, uint sz)
{
  char *mem;

  if(sz >= PGSIZE)
    panic("inituvm: more than a page");
  mem = kalloc();
  memset(mem, 0, PGSIZE);
  mappages(pgdir, 0, PGSIZE, v2p(mem), UVMPDXATTR, UVMPTXATTR);
  memmove(mem, init, sz);
}
```

Since `inituvm()` is a dedicated function for setting up the memory space for the first process which is very small, it only allocates one page for the process memory space and then copies `initcode` to the page with `memmove()`. You may notice the page is filled with 0 after `kalloc()` using `memset()`. This is a standard practice in xv6 in order to avoid any possible side effect caused by the junk data left in the page. We will not mention this practice again when similar code is encountered in the rest of the guide book.

The essential work of setting up the page table entries is done by `mappages(pgdir, 0, PGSIZE, v2p(mem), UVM`. The first parameter is the page directory table, the second parameter is the start of the mapped virtual address which is 0 for the first process, the third parameter is the size of the mapped range, the fourth parameter is the address of the physical page that the virtual address range is mapped to, and the final two parameters are the attributes for the user-space first-level and second-level page table entries. These attributes basically allow the user mode to read and write the page and describe the cache behaviour of the page etc. For more detail of the attributes, refer to Chapter 9 The Memory Management Unit of ARM Cortex-A Series Programmer's Guide.

In `mappages()`, it first gets the base address of the first and the last page of the process memory space range that is to be mapped and put them into `a` and `last` as below.

```
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, uint l1attr, uint l2attr)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
```

If the mapping is using the large page size 1 MB, the attributes of the first-level page table entries, i.e., `l1attr`, combined with the base address of the physical page `pa`, are set into the corresponding page directory entry of `pgdir` page by page until the last page (see code below).

```
if((SECTION & l1attr) != 0){// for 1 MB pages
      for(;;){
          if(a > last) break;
          if((uint)pgdir[PDX(a)] != 0) panic("remap");
          pgdir[PDX(a)] = pa | l1attr;
          a += MBYTE;
          pa += MBYTE;
      }
}
```

The above `panic()` function is used for reporting fatal kernel errors and then freezing the CPU.

If the attributes indicate the use of the small page size 4 KB, the second-level page table will be retrieved (or allocated if unavailable) using `walkpgdir()` as below.

```
else if((COARSE & l1attr) != 0){// for 4kB pages
      for(;;){
          if((pte = walkpgdir(pgdir, a, l1attr, 1)) == 0)
              return -1;
          if((uint)*pte != 0) panic("remap");
          *pte = pa | l2attr;
          if(a == last) break;
          a += PGSIZE;
          pa += PGSIZE;
      }
} else panic("Unknown page attribute");
```

After the corresponding page table entry for the current virtual page is found in the second-level page table, it is set with the second-level attributes `l2attr` combined with the base address of the physical page that is to be mapped to. The entries are set one by one until the last page is mapped. In the case of the process memory space, `l2attr` basically allows the USER mode to read and write and makes the page cacheable etc. For more details of the second-level attributes, refer to Chapter 9 The Memory Management Unit of ARM Cortex-A Series Programmer's Guide.

Note that `mappages()` assumes the physical pages starting at `pa` are contiguous if multiple pages are mapped.

Now that we know `mappages()` in detail, we will have a look at `walkpgdir()` below.

```
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, uint l1attr, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if((uint)*pde != 0){
    pgtab = (pte_t*)p2v(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    memset(pgtab, 0, PGSIZE);
    *pde = v2p(pgtab) | l1attr;
  }
  return &pgtab[PTX(va)];
```

}

In the 32-bit virtual address in ARM MMU, the most significant 10 bits are used to find the page directory entry in the first-level page table since each page directory entry governs 1 MB memory space in the virtual address space. In the case of the second-level page tables being used, the middle 8 bits (11th to 18th places) in the address from the left (the most significant side) are used to find the page table entry in the second-level page table if the page size is 4 KB. So there are 256 entries in the second-level page table corresponding to all the possible values of the 8-bit number. Finally the least significant 12 bits on the right of the address represent the offset of the address inside the 4-KB page and are not used in the page tables. However, the MMU will use those 12 bits when translating a virtual address into a physical address. The translation simply adds the 12-bit offset into the base address of the physical page found in the page table, the result of which is the physical address to be used by the MMU to access the memory.

In `walkpgdir()`, `PDX(va)` returns the value of the 10 significant bits of `va`, so `pde = &pgdir[PDX(va)]` puts the pointer of the right page directory entry of `va` into `pde`. If the entry is not empty, `pgtab = (pte_t*)p2v(PTE_` returns the base address of the second-level page table and stores it in `pgtab`. The physical address of the second-level page table is stored in the page directory entry along with the attributes. `PTE\_ADDR` is used to extract the bits of the physical address (which are the most significant 20 bits of `*pde`). Since the kernel code cannot handle physical address directly, it is converted to virtual address using `p2v`. You can find the definition of `p2v`, `PDX` etc in the header files *m*emlayout.h and *m*mu.h.

If the entry at `pde` is empty, it means the second-level page table for this entry is not allocated yet. So `walkpgdir()` allocates a page for the second-level page table and sets the page directory entry at `pde` with the attributes `l1attr` and the physical address of the second-level page table (pointed by `pgtab`).

Finally `walkpgdir()` returns `&pgtab[PTX(va)]` as the pointer to the right entry of the second-level page table for the virtual address `va`, where `PTX(va)` returns the value of the middle 8 bits of `va` used for indexing the second-level page table.

In the case of the first process, since `va` is 0, `walkpgdir()` will find the first entry in the page directory table. Since the entry is initially empty, a page for the second-level page table will be allocated and pointed by the first page directory entry. Finally the address of the first entry in the second-level page table is returned.

Now we know how the page tables and memory space are created for processes especially the first process. Below we will look at the code of `fork()` and `exec()` to understand how the process memory space is created for the rest of the processes in the system, as all other processes are derived from the first process using `fork()` and/or `exec()`.

The following code of `fork()` is very similar to `userinit()` except `copyuvm()` instead of `inituvm()` is used to create the process memory space. Other differences are that the fields of the PCB of the new process are copied from the current process including the opened files and the current working directory. We will look at the functions `filedup()` and `idup()` which duplicate the opened file table and inode (file indexing data structure) in Chapter 7.

```
int fork(void)
{
  int i, pid;
  struct proc *np;

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

  // Copy process state from p.
  if((np->pgdir = copyuvm(curr_proc->pgdir, curr_proc->sz)) == 0){
    kfree(np->kstack);
```

```
  np->kstack = 0;
  np->state = UNUSED;
  return -1;
}
np->sz = curr_proc->sz;
np->parent = curr_proc;
*np->tf = *curr_proc->tf;

// Clear r0 so that fork returns 0 in the child.
np->tf->r0 = 0;

for(i = 0; i < NOFILE; i++)
  if(curr_proc->ofile[i])
    np->ofile[i] = filedup(curr_proc->ofile[i]);
np->cwd = idup(curr_proc->cwd);

  pid = np->pid;
  np->state = RUNNABLE;
  safestrcpy(np->name, curr_proc->name, sizeof(curr_proc->name));
  return pid;
}
```

One subtle detail in the above code is that, for the new child process, `fork()` should have the return value of 0. So the register `r0` of the trap frame, where the return value is stored for system calls, is set to 0 (`np->tf->r0 = 0`). The rest of the trap frame for the child process are set to be exactly the same as the current process, as both the child and parent processes will continue the execution after the `fork()` system call though with different return values. Therefore, `fork()` is often said "called once return twice".

`copyuvm()` creates for the child process the page tables that map the same process virtual memory space as the current (parent) process calling `fork()`, but to different physical pages.

```
pde_t* copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;
  char *mem;

  if((d = setupkvm()) == 0)
    return 0;
  for(i = 0; i < sz; i += PGSIZE){
if((pte = walkpgdir(pgdir, (void *) i, UVMPDXATTR, 0)) == 0)
  panic("copyuvm: pte should exist");
if((uint)*pte == 0)
  panic("copyuvm: page not present");
pa = PTE_ADDR(*pte);
flags = PTE_FLAGS(*pte);
if((mem = kalloc()) == 0)
  goto bad;
memmove(mem, (char*)p2v(pa), PGSIZE);
if(mappages(d, (void*)i, PGSIZE, v2p(mem), UVMPDXATTR, flags) < 0)
  goto bad;
```

```
  }
  return d;

bad:
  freevm(d);
  return 0;
}
```

The above code of `copyuvm()` first allocates a page for the page directory table using `setupkvm()`. With the empty page directory table `d`, `copyuvm()` populates the entries one by one by duplicating the parent's page directory table `pgdir`. Using `walkpgdir()`, it finds the second-level page table entry `pte` for each page in the parent process memory space. The content of the page (code and data of the parent process in `pa`) is copied to a new page (`mem`) with `memmove()`. The attributes (`flags`) of the entry (`pte`) is duplicated in the page directory table `d` using `mappages()` which generates a new second-level page table entry pointing to the new page `mem` to be used by the child process. This duplication is repeated until the whole parent process memory space and its page table entries are duplicated for the child process.

As you may recall, the first process (*i*nitcode.S) immediately execute another program with `exec()`. This system call has a lot similarity to `userinit()` like creating a memory space for the new program and set up the trap frame for user-space execution, except that `exec()` has to read the program code from the file system while `userinit()` simply copies `initcode` from the kernel image in the memory. Below is the step-by-step explanation of `exec()`.

First, load the program code from the file system.

```
int exec(char *path, char **argv)
{
  char *last;
  int i, off;
  uint argc, sz, sp, ustack[3+MAXARG+1];
  struct elfhdr elf;
  struct inode *ip;
  struct proghdr ph;
  pde_t *pgdir, *oldpgdir;

  if((ip = namei(path)) == 0)
    return -1;
  ilock(ip);
  pgdir = 0;

  // Check ELF header
  if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
    goto bad;
  if(elf.magic != ELF_MAGIC)
    goto bad;
```

The above code basically finds the index node `ip` for the executable file `path`. Then the executable file header is read into a memory buffer `elf`. The executable file is in the format of ELF (Executable and Linkable Format). You can find a lot information of ELF from Google. Here is a document you may find helpful: *T*ool Interface Standard (TIS) Executable and Linking Format (ELF) Specification.

`namei()` finds the index node (inode) and `readi` reads the data of the file using the index node. To avoid data race, the index node is locked with `ilock()` before being accessed. We will explain them in detail in Chapter 7.

We leave the error handling code for you to study yourself as it does not affect much the understanding of operating systems. However, it is very important for robust kernel programming.

Then, we allocate an empty page directory table `pgdir`.

```
if((pgdir = setupkvm()) == 0)
  goto bad;
```

Next, the ELF file loadable code/data sections are read from the file and copied to the new memory space defined by `pgdir`.

```
// Load program into memory.
sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
  if(ph.type != ELF_PROG_LOAD)
    continue;
  if(ph.memsz < ph.filesz)
    goto bad;
  if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;
  if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
}
```

The information of the code sections is in the ELF header `elf`. For each code/data section, `allocuvm()` is used to create the corresponding page table entries in the memory space represented by `pgdir`. The content of the code/data section is copied to the memory space by `loaduvm()`. We will look at `allocuvm()` and `loaduvm()` shortly after `exec()` is finished.

After the code/data sections, `exec()` creates the stack space in the memory space.

```
sz = PGROUNDUP(sz);
if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
  goto bad;
clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
sp = sz;
```

The above code first creates a space of two pages using `allocuvm()`. Then it deletes the entry for the first page using `clearpteu()`. That means the stack space is limited to one page. If it overflows, page fault will happen. This is to protect the code/data sections below the stack if it overflows, as the stack grows downward.

Next, the stack should be prepared with arguments for execution of the program. As we may know that, in C program, the execution starts from the function `main()` which can have two arguments `argc` and `argv`. `argc` means how many parameters are passed in `argv` which is an array of pointers to the parameters. `argv[0]` has the name of the executable file and the rest of the array have the parameters passed from the command line (via the command shell) to the program (i.e. `main()`.

```
for(argc = 0; argv[argc]; argc++) {
  if(argc >= MAXARG)
    goto bad;
  sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
  if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
    goto bad;
```

```
    ustack[3+argc] = sp;
  }
  ustack[3+argc] = 0;
```

The above code stores the argument strings pointed by `argv[]` to the stack but prepares the rest of the stack in the temporary buffer `ustack`. The pointers to the argument strings on the stack are stored in `ustack` temporarily but will be copied to the stack later. The function `copyout()` allows the data copying to a memory space represented by `pgdir` that is not the current page table. It is essential to understand that `exec()` involves two memory spaces. One is the space of the current process calling `exec()`, and the other is the new memory space `exec()` is building for the new program. We are only allowed to access the current memory space directly with the pointers to the current space. However, to access a non-current space represented by a page directory table, e.g. the stack space pointed by `sp` in the above code, we have to use functions like `copyout()` which will be explained later in this chapter.

Then we prepare the stack in `ustack` following the C function calling convention as below.

```
  ustack[0] = 0xffffffff;  // fake return PC
  ustack[1] = argc;
  ustack[2] = sp - (argc+1)*4;  // argv pointer

  sp -= (3+argc+1) * 4;
  if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
    goto bad;
```

According to the calling convention for the function `main()`, from the bottom up on the stack, the first one is the return address for `main()`. It is set a fake return address since `main()` will never return (a system call `exit()` is always called at the end of `main()` which terminates the porgram). Then the first argument `argc` is placed followed by the pointer to `argv[]`. Finally, `copyout()` is used to copy `ustack` to the real stack in the new memory space. Note that the above deployment of arguments has not much impact on how `main()` will be invoked since the invocation of `main()` is eventually decided by the following trap frame.

Next, the fields of the current process PCB are changed accordingly.

```
  last = argv[0];
  safestrcpy(curr_proc->name, last, sizeof(curr_proc->name));

  oldpgdir = curr_proc->pgdir;
  curr_proc->pgdir = pgdir;
  curr_proc->sz = sz;
  curr_proc->tf->pc = elf.entry;  // main
  curr_proc->tf->sp = sp;
  curr_proc->tf->r0 = ustack[1];
  curr_proc->tf->r1 = ustack[2];
```

Among other fields, the trap frame is the key one to make things right. The first instruction is set to `elf.entry` which points to `main()`. The stack pointer is set to the current top `sp`. The registers `r0` and `r1`, which contain the arguments according to ARM calling convention, point to `argc` and `argv`. So, when the process is back to the user space, `main()` with arguments `argc` and `argv` will be executed.

The final key step is to switch to the new memory space and remove the old memory space as below.

```
  switchuvm(curr_proc);
  freevm(oldpgdir);
  return 0;
```

`switchuvm()` replace the current memory space with the new space by changing the entries of the current page directory table used for the user process space (0-1 GB).

```
void switchuvm(struct proc *p)
{
  pushcli();
  if(p->pgdir == 0)
    panic("switchuvm: no pgdir");
  memmove((void *)kpgdir, (void *)p->pgdir, PGSIZE);  // switch to new user address space
  flush_idcache();
  flush_tlb();
  popcli();
}
```

The above code replaces the first page of the current page directory table (pointed by `kpgdir`) with the new entries in `p->pgdir`. Since the virtual memory space is going to change, we need to flush the instruction and data caches and the TLB cache in `switchuvm()`.

`freevm()` frees all pages used by a memory space and its page tables.

```
void freevm(pde_t *pgdir)
{
  uint i;

  if(pgdir == 0)
    panic("freevm: no pgdir");
  deallocuvm(pgdir, USERBOUND, 0);
  for(i = 0; i < NPDENTRIES; i++){
    if((uint)pgdir[i] != 0){
      char * v = p2v(PTE_ADDR(pgdir[i]));
      kfree(v);
    }
  }
  kfree((char*)pgdir);
}
```

In the above code, `deallocuvm()` reduces the user memory space to size 0 by clearing the page table entries and freeing the pages allocated to the user space. Its code is shown as below.

```
int deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
pte_t *pte;
uint a, pa;

if(newsz >= oldsz)
  return oldsz;

a = PGROUNDUP(newsz);
for(; a  < oldsz; a += PGSIZE){
  pte = walkpgdir(pgdir, (char*)a, UVMPDXATTR, 0);
  if(!pte)
    a += (NPTENTRIES - 1) * PGSIZE;
  else if(*pte != 0){
```

```
      pa = PTE_ADDR(*pte);
      if(pa == 0)
        panic("kfree");
      char *v = p2v(pa);
      kfree(v);
      *pte = 0;
    }
  }
  return newsz;
}
```

The above code walks through the second-level page tables. For each second-level page table entry in the address range between `newsz` and `oldsz`, the mapped physical page is freed (`kfree(v)`) and the entry is cleared (`*pte = 0`).

Now we look at other functions handling the process memory space.

First, let us look at the simple `clearpteu()` function shown below.

```
void clearpteu(pde_t *pgdir, char *uva)
{
  pte_t *pte;

  pte = walkpgdir(pgdir, uva, UVMPDXATTR, 0);
  if(pte == 0)
    panic("clearpteu");
  *pte &= ~PTX_AP(U_AP);
}
```

The above code, walking through the page table entries, finds the second-level page table entry for the address `uva` and clears the attributes of the entry so the page at `uva` is not accessible anymore. If it is accessed, a page fault will happen. In xv6, if a page fault happens on the user memory space, the process will simply be killed. So, in case of stack overflow, the process is killed.

`allocuvm()` is the reverse operation of `deallocuvm()`. The code is shown below.

```
int allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  char *mem;
  uint a;

  if(newsz >= USERBOUND)
    return 0;
  if(newsz < oldsz)
    return oldsz;

a = PGROUNDUP(oldsz);
for(; a < newsz; a += PGSIZE){
  mem = kalloc();
  if(mem == 0){
    cprintf("allocuvm␣out␣of␣memory\n");
    deallocuvm(pgdir, newsz, oldsz);
    return 0;
  }
  memset(mem, 0, PGSIZE);
```

```
    mappages(pgdir, (char*)a, PGSIZE, v2p(mem), UVMPDXATTR, UVMPTXATTR);
  }
  return newsz;
}
```

The function extends the memory space represented by `pgdir` from `oldsz` to `newsz`. For each second-level page table entry in the range between `oldsz` to `newsz`, the function gets a free page `mem` and maps the corresponding virtual address `a` to the physical address of `mem` with the proper attributes using `mappages()`.

`loaduvm()` loads a program section from an ELF file into a process memory space. The code is shown as below.

```
int loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
  uint i, pa, n;
  pte_t *pte;

  if((uint) addr % PGSIZE != 0)
    panic("loaduvm:␣addr␣must␣be␣page␣aligned");
  if((uint)addr + sz > USERBOUND)
    panic("loaduvm:␣user␣address␣space␣exceeds␣the␣allowed␣space␣(>␣0x80000000)");
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, addr+i, UVMPDXATTR, 0)) == 0)
      panic("loaduvm:␣address␣should␣exist");
    pa = PTE_ADDR(*pte);
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    if(readi(ip, p2v(pa), offset+i, n) != n)
      return -1;
  }
  return 0;
}
```

The above function assumes the page table entries of the memory space represented by `pgdir` are set up by `allocuvm()`. Now it just needs to copy the code/data section from the ELF file (pointed by `ip`) at `offset` with size `sz` to the address `addr` of the memory space. It copies the ELF section page by page through finding each second-level page table entry and the address of the mapped page. The content is copied to the page `pa` by `readi()`.

Note that we don't use the physical address `pa` of the page directly. The address is converted to the kernel virtual address for `readi()` using `p2v()`. The following function `copyout()` has similar conversion except it converts a user-space address to a kernel-space address.

```
int copyout(pde_t *pgdir, uint va, void *p, uint len)
{
  char *buf, *pa0;
  uint n, va0;

  buf = (char*)p;
  while(len > 0){
    va0 = (uint)PGROUNDDOWN(va);
```

```
    pa0 = uva2ka(pgdir, (char*)va0);
    if(pa0 == 0)
      return -1;
    n = PGSIZE - (va - va0);
    if(n > len)
      n = len;
    memmove(pa0 + (va - va0), buf, n);
    len -= n;
    buf += n;
    va = va0 + PGSIZE;
  }
  return 0;
}
```

The above function is very useful to copy data from the current memory to a non-current memory space represented by the page directory table `pgdir`. The source of the data in the current memory space is pointed by `p` with length `len`. The destination for the data is pointed by `va` but in the non-current memory space represented by `pgdir`. The key of the function is to find the corresponding address in the kernel space (i.e. the current memory space) for `va` using `uva2ka()`. The rest of the function is just a routine page-by-page copying.

Below is the code for `uva2ka()`.

```
char* uva2ka(pde_t *pgdir, char *uva)
{
  pte_t *pte;

  pte = walkpgdir(pgdir, uva, UVMPDXATTR, 0);
  if((uint)*pte == 0)
    return 0;
  if(((uint)*pte & PTX_AP(U_AP)) == 0)
    return 0;
  return (char*)p2v(PTE_ADDR(*pte));
}
```

`uva2ka()`, walking through the page tables of the memory space `pgdir`, finds the corresponding second-level page table entry `pte` for the virtual address `uva` in the non-current memory space. Then it extracts the physical address of the mapped page from `pte`. Finally it converts the physical address into its kernel-space address using `p2v()` and returns. The key to understand the conversion is that the same physical page can be mapped into multiple memory spaces. It is at least mapped to the kernel space in the boot procedure. It could be mapped to one or more process memory spaces. To access an address in a non-current process memory space, we should find its physical address via its page tables and then convert the physical address to its kernel address (via `p2v()`).

So far we have gone through all functions in *v*m.c that are used to manipulate the virtual memory space of a process. Hope you have got a better understanding of how xv6 manages the process memory space.

Chapter 4

System Calls and Interrupts

System Calls

System calls are collectively an interface for user programs (processes) to request service from the OS kernel. xv6 has the following system calls.

- fork() Create a new process
- exit() Terminate the current process
- wait() Wait for a child process to exit
- kill(pid) Terminate a process identified with *p*id
- getpid() Return the ID number of the current process
- sleep(n) Put the current process into sleep for *n* seconds
- exec(filename, *argv) Load a file named *f*ilename and execute it with arguments in *a*rgv
- sbrk(n) Update the process memory space by *n* bytes
- open(filename, flags) Open a file named *f*ilename with *f*lags indicating read/write
- read(fd, buf, n) Read *n* bytes from an open file *f*d into *b*uf
- write(fd, buf, n) Write *n* bytes from *b*uf to an open file *f*d
- close(fd) Close the open file *f*d
- dup(fd) Duplicate the open file *f*d into another file descriptor (the return value)
- pipe(p) Create a pipe and return the file descriptors in *p*
- chdir(dirname) Change the current working directory of the process to *d*irname
- mkdir(dirname) Create a new directory *d*irname
- mknod(name, major, minor) Create a device file *n*ame with *m*ajor and *m*inor numbers
- fstat(fd) Return status information about an open file *f*d
- link(f1, f2) Create another name *f*2 for the file *f*1

- unlink(filename) Remove a file named *f*ilename

A sample assembly code for making system calls (e.g. `fork()`) in RPI2 is as below. All system calls follow the same convention except the system call number (e.g. `SYS_fork` in the code below) is different.

```
.globl fork
fork:
    push {lr}
    push {r3}
    push {r2}
    push {r1}
    push {r0}
    mov r0, #SYS_fork
    swi #T_SYSCALL
    pop {r1} /* to avoid overwrite of r0 */
    pop {r1}
    pop {r2}
    pop {r3}
    pop {lr}
    bx lr
```

The above code is an interface to system call `fork()` in user mode. Similar code for other system calls can be found from *u*progs/usys.S.

When a user process calls function `fork()`, the above code will be executed. When a system call is invoked, the parameters for a system call are passed through the registers `r0`, `r1`, `r2`, and `r3` to the above assembly function. The parameters are then pushed to the user stack for later access by the kernel code. The `swi` instruction switches to the kernel (SVC) mode and the system call handler in the xv6 kernel will be executed. After the system call returns from the kernel mode, the stack space for the assembly function is cleared but the return value of the system call is kept in `r0` before returning to the caller function.

When the `swi` instruction switches to the kernel (SVC) mode, the hardware will invoke the system call handler according to the following setting of the trap vector in source/exception.S.

```
vectors:
        ldr pc, reset_handler
        ldr pc, undefintr_handler
        ldr pc, swi_handler
        ldr pc, prefetch_handler
        ldr pc, data_handler
        ldr pc, unused_handler
        ldr pc, irq_handler
        ldr pc, fiq_handler
reset_handler:
        .word hang    /* reset, in svc mode already */
undefintr_handler:
        .word do_und      /* undefined instruction */
swi_handler:
        .word do_svc    /* SWI & SVC */
prefetch_handler:
        .word do_pabt      /* prefetch abort */
data_handler:
        .word do_dabt     /* data abort */
unused_handler:
```

```
        .word hang      /* reserved */
irq_handler:
        .word do_irq    /* IRQ */
fiq_handler:
        .word hang    /* FIQ */
```

For system calls, the address of `do_svc` is loaded into `pc` and the following code is executed. It is worth noting that switching to the SVC mode involves the switching to the kernel stack space, which is set for each process when the process is scheduled to run (refer to the `swtch()` function in *e*xception.S and Chapter 6).

```
do_svc:
        push {lr}
        mrs lr, spsr
        push {lr}
        mrs lr, cpsr
        push {lr}
        mrc p15, 0, lr, c6, c0, 2  /* read Instruction Fault Address Register (IFAR) */
        push {lr}
        mov lr, #0x40
        push {lr}
        STMFD sp, {r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14}
        sub sp, sp, #60
        mov r0, sp   /* save sp */
        STMFD r0, {r13}^   /* save user mode sp */
        mov r1, r1 /* three nops after STM with user mode banked registers */
        mov r1, r1
        mov r1, r1
        mov sp, r0  /* restore sp */
        sub sp, sp, #4
        mov r0, sp
        bl trap

.global trapret
trapret:
        mov r0, sp /* save sp in case it is changed to sp_usr after the following LDMFD instruction */
        LDMFD r0, {r13}^ /* restore user mode sp */
        mov r1, r1  /* three nops after LDMFD */
        mov r1, r1
        mov r1, r1
        mov sp, r0  /* restore sp */
        add sp, sp, #4
        LDMFD sp, {r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12}
        add sp, sp, #72
        pop {lr}
        msr spsr, lr
        pop {lr}
        movs pc, lr  /* subs pc,lr,#0 */
```

The code above pushes the values of the registers to the kernel stack of the current process according to `struct trapframe` in *i*nclude/arm.h, as presented in Chapter 2. Then the following function `trap()` is called, followed by the restoration of the registers and return of user mode at `trapret`.

```
void trap(struct trapframe *tf)
{
        intctrlregs *ip;
        uint istimer;

  if(tf->trapno == T_SYSCALL){
    if(curr_proc->killed)
      exit();
    curr_proc->tf = tf;
    syscall();
    if(curr_proc->killed)
      exit();
    return;
  }

  istimer = 0;
switch(tf->trapno){
case T_IRQ:
      ip = (intctrlregs *)INT_REGS_BASE;
      while(ip->gpupending[0] || ip->gpupending[1] || ip->armpending){
          if(ip->gpupending[0] & (1 << IRQ_TIMER3)) {
              istimer = 1;
              timer3intr();
          }
          if(ip->gpupending[0] & (1 << IRQ_MINIUART)) {
              miniuartintr();
          }
      }

      break;
default:
  if(curr_proc == 0 || (tf->spsr & 0xF) != USER_MODE){
    // In kernel, it must be our mistake.
    cprintf("unexpected trap %d from cpu %d addr %x spsr %x cpsr %x ifar %x\n",
            tf->trapno, curr_cpu->id, tf->pc, tf->spsr, tf->cpsr, tf->ifar);
    panic("trap");
  }
  // In user space, assume process misbehaved.
  cprintf("pid %d %s: trap %d on cpu %d "
          "addr 0x%x spsr 0x%x cpsr 0x%x ifar 0x%x--kill proc\n",
          curr_proc->pid, curr_proc->name, tf->trapno, curr_cpu->id, tf->pc,
          tf->spsr, tf->cpsr, tf->ifar);
  curr_proc->killed = 1;
}

// Force process exit if it has been killed and is in user space.
// (If it is still executing in the kernel, let it keep running
// until it gets to the regular system call return.)

if(curr_proc){
```

```
        if(curr_proc->killed && (tf->spsr&0xF) == USER_MODE)
                exit();

  // Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to check nlock.
        if(curr_proc->state == RUNNING && istimer)
                yield();

  // Check if the process has been killed since we yielded
        if(curr_proc->killed && (tf->spsr&0xF) == USER_MODE)
                exit();
  }
}
```

The `trap()` function deals with system calls, interrupts, and other exceptions. We focus on system call here.

For a system call (trap number set to `T_SYSCALL`), the function `syscall()` (shown below) is invoked. Since it is possible that the current process has been killed before and after `syscall()`, `curr_proc->killed` is checked and if true `exit()` is called instead to terminate the process, which is equivalent to the system call `exit()`.

```
void syscall(void)
{
  int num;

  num = curr_proc->tf->r0;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {

    if(num == SYS_exec) {
        if(syscalls[num]() == -1) curr_proc->tf->r0 = -1;
    } else curr_proc->tf->r0 = syscalls[num]();
  } else {
    cprintf("%d␣%s:␣unknown␣sys␣call␣%d\n",
            curr_proc->pid, curr_proc->name, num);
    curr_proc->tf->r0 = -1;
  }
}
```

In `syscall()`, according to the system call number, the corresponding function `syscalls[num]()` for the system call is executed. The return value of a syscall function `syscalls[num]()` is stored in `r0` of the process' `trapframe`. The only exception for processing the return value of a syscall function is `exec()`, which should never return unless there is an error, in which case -1 is returned via `r0`. Below is a list of functions for the system calls.

```
static int (*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,
};
```

In the case of `fork()`, `sys_fork()` in *sys*proc.c is executed, which calls `fork()` in *p*roc.c directly. As mentioned before, `fork()` is very similar to `userinit()`, though with some slight differences. For those interested, it is worthwhile to study and compare the two functions in order to understand how a new process is created.

Interrupts

Interrupt handling is very similar to handling system calls. The only complication is that ARM adopts different modes (IRQ and FIQ) for interrupts. FIQ is for fast processing one interrupt while IRQ is for all other interrupts. Both modes have their own stack spaces different from the SVC mode. Since xv6 assumes a single priviledged mode, we have to switch to the SVC mode from IRQ and FIQ in order to avoid extensive modification of the `trap()` function. Note that we do not use FIQ yet as it is often used for USB devices but we do not support them yet.

```
do_irq:
        STMFD sp, {r0-r4}
        mov r0, #0x80
        b _switchtosvc
_switchtosvc:
        mrs r1, spsr
        sub r2, lr, #4
        mov r3, sp
        mrs lr, cpsr
        bic lr, #0x0000001F /* PSR_MASK */
        orr lr, #0x00000080 /* PSR_DISABLE_IRQ */
        orr lr, #0x00000013 /* PSR_MODE_SVC */
        msr cpsr, lr /* switch to svc */
        push {r2}
        push {r1}
        mrs r1, cpsr
        push {r1}
        mrc p15, 0, r1, c6, c0, 2  /* read Instruction Fault Address Register (I
FAR) */
        push {r1}
        push {r0}
```

```
sub r1, r3, #20
LDMFD r1, {r0-r4}
STMFD sp, {r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14}
sub sp, sp, #60
mov r0, sp   /* save sp */
STMFD r0, {r13}^    /* save user mode sp */
mov r1, r1 /* three nops after STM with user mode banked registers */
mov r1, r1
mov r1, r1
mov sp, r0  /* restore sp */
sub sp, sp, #4
mov r0, sp

bl trap

mov r0, sp
        add r0, #76
        LDMIA r0, {r1}
        mov r2, r1
        and r2, #0xf
        cmp r2, #0
        beq _backtouser
        msr cpsr, r1
        add sp, #4
        LDMFD sp, {r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12}
        add sp, sp, #56
        pop {r14}
        add sp, sp, #16
        pop {pc}

_backtouser:
        mov r0, sp /* save sp in case it is changed to sp_usr after the following LDMFD instruction */
        LDMFD r0, {r13}^ /* restore user mode sp */
        mov r1, r1  /* three nops after LDMFD */
        mov r1, r1
        mov r1, r1
        mov sp, r0  /* restore sp */
        add sp, sp, #4
        LDMIA sp, {r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12}
        add sp, sp, #72
        pop {lr}
        msr spsr, lr
        pop {lr}
        movs pc, lr  /* subs pc,lr,#0 */
```

In the above code of *e*xception.S, since it is in the IRQ mode, we need to switch to the SVC mode first. First, the values of `r0`, `r1`, `r2`, and `r3` are stored in the stack space of the IRQ mode. Then, set `r0` to 0x80, which will tell `trap()` that it is an IRQ event instead of say system call. Next, the code switches to `_switchtosvc`.

To switch to SVC, we first save the previous status register `spsr`, the return address register `lr`, and

the stack pointer `sp` into `r1`, `r2`, and `r3`, as they will change once switching to SVC. Then, we move the current value of `cpsr` to `lr` and set the status value properly (clear the PSR bits, disable IRQ, and set the PSR bits to SVC mode). Next, we switch to SVC by setting `cpsr` to the value of `lr`.

Note that, when `lr` is saved to `r2`, there is a subtle modification of `lr` by `sub r2, lr, #4`. Usually `lr` contains the code address interrupted by IRQ. However, since ARM architectures add 8 to the interrupted address, we subtract 4 from `lr`, which makes `lr` point exactly to the address to be returned from IRQ.

After switching to SVC, similar to `do_svc`, the values of the registers are pushed to the kernel (SVC) stack of the current (interrupted) process or the scheduler, according to the layout of `trapframe`. The first one is the return address, which is now in `r2`. The second one is the previous processor status which is now in `r1`. Then, the current processor status `cpsr`, the IFAR register, the event number (in `r0`) for `trap()` are pushed.

Next, we push the general registers `r0` to `r14` to the stack. However, since the original values of `r0` to `r3` from the interrupted context are stored in the stack of the IRQ mode, we now restore them with `LDMFD r1, {r0-r4}`, where `r1` is now pointing to the stack address where `r0-r4` are stored. After the restoration of `r0-r4`, all the registers are pushed to the stack with `STMFD`.

Finally, we adjust the current stack pointer accordingly and then save the current stack pointer of the user mode. The instruction `STMFD r0, {r13}^` takes longer time due to its access to the registers of a different mode, so we have to use three nops to make sure the instruction is complete. Then, the user stack pointer is stored to the kernel stack before `trap()` is called.

After `trap()` is returned, all corresponsing registers are restored accordingly if the previous processor status is in SVC mode. However, there is a slight complication if the previous processor status is in user mode. This case is dealt with at `_backtouser`. The stack pointer at the user mode is restored with `LDMFD r0, {r13}^` before all other registers are restored. Lastly, `spsr` is restored and `pc` is set to the return address in user mode with `movs pc, lr`. Since the stack pointer of the user mode is already set, once the user mode is returned, the value of the current `sp` is irrelevant. Note that we cannot use `pop` to set `pc` in a different mode, in which case `pc` should be set with `movs`.

In `trap()`, the IRQ events are dealt with as below.

```
switch(tf->trapno){
case T_IRQ:
      ip = (intctrlregs *)INT_REGS_BASE;
      while(ip->gpupending[0] || ip->gpupending[1] || ip->armpending){
          if(ip->gpupending[0] & (1 << IRQ_TIMER3)) {
              istimer = 1;
              timer3intr();
          }
          if(ip->gpupending[0] & (1 << IRQ_MINIUART)) {
              miniuartintr();
          }

      break;
```

When an interrupt occurs, a bit in `gpupending` or `armpending` will be set accordingly. So far we only handle two interrupts: timer and uart. For the timer interrupt, `timer3intr()` is the handler, while `miniuartintr()` handles the uart interrupt. These functions will be explained in the following Chapter 5.

For other exceptions like undefined instructions, data abort, etc, `trap()` will either send panic message if the current code is in the kernel space, or kill the process if it is caused by user code. To save you the trouble of turning back pages, the related code is shown as below.

```
default:
```

```
if(curr_proc == 0 || (tf->spsr & 0xF) != USER_MODE){
  // In kernel, it must be our mistake.
  cprintf("unexpected trap %d from cpu %d addr %x spsr %x cpsr %x ifar %x\n",
          tf->trapno, curr_cpu->id, tf->pc, tf->spsr, tf->cpsr, tf->ifar);
  panic("trap");
}
// In user space, assume process misbehaved.
cprintf("pid %d %s: trap %d on cpu %d "
        "addr 0x%x spsr 0x%x cpsr 0x%x ifar 0x%x--kill proc\n",
        curr_proc->pid, curr_proc->name, tf->trapno, curr_cpu->id, tf->pc,
        tf->spsr, tf->cpsr, tf->ifar);
curr_proc->killed = 1;
```

After handling the the timer interrupt with `timer3intr()`, we need to schedule the current process out from the CPU by calling `yield()` (scheduling functions like `yield()` will be discussed in Chapter 6). Note that, before yielding the CPU, we should make sure the process does not hold any locks. More details will be discussed in Chapter 8.

Also if the current process has been sent a *k*illed signal and is in user mode, terminate the process with `exit()`. However, if the process is executing in the kernel (SVC) mode, let it keep running until the current system call returns, by which time it will be terminated anyway. The code is shown as below.

```
if(curr_proc){
      if(curr_proc->killed && (tf->spsr&0xF) == USER_MODE)
              exit();

// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
      if(curr_proc->state == RUNNING && istimer)
              yield();

// Check if the process has been killed since we yielded
      if(curr_proc->killed && (tf->spsr&0xF) == USER_MODE)
              exit();
```

Chapter 5

Device Drivers

A device driver is a piece of code in OS kernel that handles a particular device. A device driver usually has an interrupt handler for processing the interrupt requests from the device. It also has other utilities that read from or write to or set control status of the device. In OS like Linux, device drivers are a major part of the kernel, though many of them are installed as separate modules. In xv6, we only have drivers for three devices: system clock (aka. timer), uart and a ramdisk. We will have a look at their code in detail.

System Clock

The system clock in Raspberry Pi has a 64-bit free running counter. It resides in the GPU. The clock runs at 1 MHz, which means the clock ticks every microsecond. And the free running counter increases at every tick.

The clock provides four timers (aka. channels) for use. Each timer has 32-bit compare register, which is compared against the 32 least significant bits of the free running counters. If the two 32-bit values match, the system clock generates an interrupt for the corresponding timer. Basically the driver for the system clock could handle four timers. But in our driver code, we only handle timer 3, the last timer, as xv6 only needs one timer so far for the scheduler.

The following code in source/timer.c shows how the clock is initialized.

```
void timer3init(void)
{
uint v;

        enabletimer3irq();
        v = inw(TIMER_REGS_BASE+COUNTER_LO);
        v += TIMER_FREQ;

        outw(TIMER_REGS_BASE+COMPARE3, v);
        ticks = 0;
}

void enabletimer3irq(void)
{
        intctrlregs *ip;
```

```
        ip = (intctrlregs *)INT_REGS_BASE;
        ip->gpuenable[0] |= 1 << IRQ_TIMER3; // enable the system timer3 irq
}
```

In `timer3init()`, it first enables the interrupt for timer 3. The GPU interrupt table described in Chapter 7 Interrupts of *B*CM2835 ARM Peripherals is very unclear regarding which bits are for the four timers. However, from guestimation and heresay from google search, it seems the bits 0, 1, 2, and 3 of the GPU IRQ enabling register are for the four timers respectively. Most importantly, it works for our timer 3. So, in `enabletimer3irq()`, we set the bit 3 (`IRQ_TIMER3` is 3) in the IRQ enabling register in order to enable the interrupt for timer 3. For more details of the control registers of the GPU interrupts, refer to `struct intctrlregs` in *i*nclude/types.h and Chapter 7 Interrupts of *B*CM2835 ARM Peripherals.

Back to `timer3init()`, the lower 32 bits of the free running counter is read (`v = inw(TIMER_REGS_BASE+COUNTER`
and then the value is increased by 10,000 (`TIMER_FREQ`) ticks, which means the timer should set off after 10 milliseconds. Finally, the timer 3 is set with the set-off time (`outw(TIMER_REGS_BASE+COMPARE3, v);`). Note that `ticks` is initialized to 0, which is the number of the timer interrupts and should not be confused with the ticks of the system clock.

Since the clock device is simple, after initialization, the driver only needs to handle the timer interrupt with the following interrupt handler `timer3intr()` invoked by `trap()` in *s*ource/trap.c.

```
void timer3intr(void)
{
uint v;
        outw(TIMER_REGS_BASE+CONTROL_STATUS, (1 << IRQ_TIMER3)); // clear timer3 irq

        ticks++;
        wakeup(&ticks);

        // reset the value of compare3
        v=inw(TIMER_REGS_BASE+COUNTER_LO);
        v += TIMER_FREQ;
        outw(TIMER_REGS_BASE+COMPARE3, v);
}
```

When the timer interrupt occurs, the corresponding IRQ status bit for the timer is cleared (`outw(TIMER_REGS_BASE`
otherwise, the interrupt handler will be called repeatedly. After `ticks` is increased by 1, the handler wakes up the processes that are sleeping on `ticks`. Usually the sleeping processes will check how many ticks they slept to decide if they should sleep again or not. For more details, refer to `sys_sleep()` in *s*ource/sysproc.c.

Finally, the timer 3 is reset as in `timer3init()` so that it will set off again in the next 10 milliseconds.

Note that, at the end of *s*ource/timer.c, there is a `delay()` function which is not relevant to the system clock. It uses the ARM timestamp to achieve a delay by busy waiting.

UART

Raspberry Pi has three names for the packaged chip: BCM2835 for Raspberry Pi 1 using ARMv6, BCM2836 for Raspberry Pi 2 using ARMv7, and BCM2837 for Raspberry Pi 3 using ARMv8. Their difference is mainly the ARM CPU version, but the peripherals on the chip are almost the same, with adjusted I/O memory addresses. So we collectively talk about the chips using the name BCM283x.

BCM283x has two UARTS: mini UART and PL011 UART. UART stands for Universal Asynchronous Receiver/Transmitter. It performs serial-to-parallel conversion on data characters received from an external peripheral device like a terminal or modem. Conversely, it performs parallel-to-serial conversion

on data characters received from the Advanced Peripheral Bus (APB) in the chip. However, in xv6, we only developed a driver for the mini UART to support the console function of xv6.

The mini UART is a low throughput UART intended to be used for a console. It needs to be enabled by correctly setting the mode of the corresponding GPIO pins (14 and 15) before it can be used. The details of the mini UART can be found from *S*ection 2.2 Mini UART of *B*CM2835 ARM Peripherals.

Below is the code for UART initialization.

```
void uartinit(void)
{
        outw(AUX_ENABLES, 1);
        outw(AUX_MU_CNTL_REG, 0);
        outw(AUX_MU_LCR_REG, 0x3);
        outw(AUX_MU_MCR_REG, 0);
        outw(AUX_MU_IER_REG, 0x1);
        outw(AUX_MU_IIR_REG, 0xC7);
        outw(AUX_MU_BAUD_REG, 270); // (250,000,000/(115200*8))-1 = 270

        setgpiofunc(14, 2); // gpio 14, alt 5
        setgpiofunc(15, 2); // gpio 15, alt 5

        outw(GPPUD, 0);
        delay(10);
        outw(GPPUDCLK0, (1 << 14) | (1 << 15) );
        delay(10);
        outw(GPPUDCLK0, 0);

        outw(AUX_MU_CNTL_REG, 3);
        enableirqminiuart();
}
```

The `AUX_ENABLES` register is used to enable three auxiliary peripherals: mini UART, SPI1 and SPI2. Since we don't use SPI devices, we only enable mini UART, which sets the least significant bit of `AUX_ENABLES`.

The `AUX_MU_CNTL_REG` register provides access to some extra features in addition to enabling the transmitter and the receiver. For example, you can set FIFO level in RTS auto-flow mode. We don't use these features, though they could be explored in an advanced UART driver. The register is initially set to 0 to temporarily disable the transmitter and the receiver, but will be set to 3 soon to enable them after other settings for the UART are done.

The `AUX_MU_LCR_REG` register controls the line data format and enables access to the baudrate register through the first two registers (`AUX_MU_IO_REG` and `AUX_MU_IER_REG`) if bit 7 of the register is set. We set the last two bits. The setting of the last bit means the UART works in 8-bit mode instead of 7-bit mode for each character. It is unclear why the second last bit should be set, but the UART would not work if it is not set. The *S*ection 2.2 Mini UART of *B*CM2835 ARM Peripherals has no clear description of the bit though.

The `AUX_MU_MCR_REG` register controls the modem signals. Since the mini UART has no modem signals, we simply set the register to 0.

The `AUX_MU_IER_REG` register is used to enable interrupts. 0x1 means enabling transmitter according to *S*ection 2.2 Mini UART of *B*CM2835 ARM Peripherals, but has really enabled the receiver interrupt. So the manual of BCM2835 is sometimes really rubbish.

The `AUX_MU_IIR_REG` register shows the interrupt status. It also allows writing the two bits (0x6) to clear the receive and transmit FIFOs. So it seems 0x6 is sufficient according to the manual. However, we

have to use 0xC7 to enable the FIFOs and clear the pending interrupt in addition to clearing the FIFOs. Again the manual is quite misleading.

The `AUX_MU_BAUD_REG` register is used to set the baudrate according to the equation in Section 2.2.1 of *B*CM2835 ARM Peripherals. In our case, we set the baudrate to 115,200.

Then we use `setgpiofunc()` to set the GPIO pins 14 and 15 to function as transmit and receive lines (TXD1 and RXD1) of the mini UART.

There are 54 general-purpose I/O (GPIO) lines in BCM2835. All pins have at least two functions, input and output, and possibly up to six alternative functions within the BCM System on Chip. For example, pins 14 and 15 can be used as TXD1 and RXD1 lines for the mini UART as well as the general I/O pins (input and output) programmable by software. The following `setgpiofunc()` function sets the function or alternative function for a GPIO pin.

```
void setgpiofunc(uint pin, uint func)
{
        uint sel, data, shift;

        if(pin > 53) return;
        sel = 0;
        while (pin > 10) {
            pin = pin - 10;
            sel++;
        }
        sel = (sel << 2) + GPFSEL0;
        data = inw(sel);
        shift = pin + (pin << 1);
        data &= ~(7 << shift);
        outw(sel, data);
        data |= func << shift;
        outw(sel, data);
}
```

GPIO has six GPFSEL registers, 32 bits each, to configure the alternative functions of the 54 pins. Each pin needs three bits to set its eight possible functions. So each 32-bit GPFSEL register controls 10 pins sequentially: GPFSEL0 for pins 0-9, GPFSEL1 for pins 10-19, GPFSEL2 for pins 20-29, and so on. The addresses of these GPFSEL registers are contiguous.

The above `setgpiofunc()` function basically find the right GPFSEL register and the right 3 bits in the register through the pin number, and then set the 3 bits with the function code `func`, without changing the other bits of the register. Below is a list of the GPIO function code:

```
000 = the GPIO Pin is an input
001 = the GPIO Pin is an output
100 = the GPIO Pin takes alternate function 0
101 = the GPIO Pin takes alternate function 1
110 = the GPIO Pin takes alternate function 2
111 = the GPIO Pin takes alternate function 3
011 = the GPIO Pin takes alternate function 4
010 = the GPIO Pin takes alternate function 5
```

In our case, the pins 14 and 15 are set the alternate function 5 (with a code 010), and so are used as TXD1 and RXD1 of the mini UART.

If a pin's function is set as output, then two 32-bit registers GPSET0 and GPSET1 are used to set the pin's output as high (1), and another two registers GPCLR0 and GPCLR1 are used to set the pin's output

as low (0). Usually one bit in the register pairs is corresponding to one of the 54 pins, though some bits are not used as there are 64 bits in total.

If a pin's function is set as input, another two 32-bit registers GPLEV0 and GPLEV1 are used to read the input voltage of the pin. If the input is high voltage, the corresponding bit of the pin is 1; otherwise, it is 0.

The two GPIO Event Detect Status Registers GPEDS0 and GPEDS1 are used to show for each pin if an event (edge event or level event) is detected. It is cleared by writing 1 to the relevant bit. The interrupt controller can be programmed to interrupt the ARM CPU when any of the status bits are set. However, the details are not clear in *B*CM2835 ARM Peripherals. I am going to reverse-engineer the Linux system on RPI 2/3 to find out the details.

An edge event is the change of voltage from low to high or high to low, the detection of which is enabled by the following Edge Detect Enable Registers. A level event is the matching of the expected voltage level set by the following Low/High Detect Enable Registers.

The two GPIO Rising Edge Detect Enable Registers GPREN0 and GPREN1 are used for each pin to enable the event detection of the voltage change from low to high. Likewise, the two GPIO Falling Edge Detect Enable Registers GPFEN0 and GPFEN1 are used for each pin to enable the event detection of the voltage change from high to low.

The two GPIO High Detect Enable Registers GPHEN0 and GPHEN1 are used for each pin to enable the event detection of a high voltage level. If the pin is still high when an attempt is made to clear the status bit in GPEDS0 or GPEDS1, the status bit will remain set. Likewise, the two GPIO Low Detect Enable Registers GPLEN0 and GPLEN1 are used for each pin to enable the event detection of a low voltage level.

There are also two GPIO Asynchronous Rising Edge Detect Enable Registers GPAREN0 and GPAREN1 and two GPIO Asynchronous Falling Edge Detect Enable Registers GPAFEN0 and GPAFEN1 which are similar to GPREN0, GPREN1, GPFEN0, and GPFEN1 but they enable the detection of voltage changes that are not synchronous with the system clock. These registers are useful to detect events from external devices that do not use the system clock.

Each GPIO pin can be pull up or down by three registers: GPIO Pull-Up/Down (GPPUD), GPIO Pull-Up/Down Clock registers (GPPUDCLK0 and GPPUDCLK1). The GPPUD is used to indicate if the following setting of the pins is pull-up, pull-down, or off (disable pull-up/down). There values for GPPUD are binary numbers 10, 01, and 00 respectively. The GPPUDCLK0 and GPPUDCLK1 registers are used to set the pull-up/down features for the individual pins. The procedure is as below.

1. Write to GPPUD to set the required control signal (pull-up, pull-down, or off) with 10, 01, or 00, respectively.
2. Wait 150 cycles to provide the required setup time for the control signal.
3. Write to GPPUDCLK0 or GPPUDCLK1 to clock the control signal into the GPIO pins you wish to set; only the pins that receive a clock will be modified while others will retain their previous state.
4. Wait 150 cycles to provide the required hold time for the control signal.
5. Write 0 to GPPUD to remove the control signal (disable pull-up/down).
6. Write 0 to GPPUDCLK0 or GPPUDCLK1 to remove the clock.

Below is an example in our UART driver to disable the pull-up/down features of pins 14 and 15 so that they could be used by mini UART as TXD1 and RXD1.

```
outw(GPPUD, 0);
delay(10);
outw(GPPUDCLK0, (1 << 14) | (1 << 15) );
```

```
        delay(10);
        outw(GPPUDCLK0, 0);

        outw(AUX_MU_CNTL_REG, 3);
        enableirqminiuart();
```

From the above code, we can see that, after setting up the pins for mini UART, we use `outw(AUX_MU_CNTL_REG, 3)` to enable the transmitter and the receiver, and then enable the delivery of the interrupt of the min UART by setting the system's interrupt enabling register as below.

```
void enableirqminiuart(void)
{
        intctrlregs *ip;

        ip = (intctrlregs *)INT_REGS_BASE;
        ip->gpuenable[0] |= (1 << 29);   // enable the miniuart through Aux
}
```

In Raspberry Pi BCM System on Chip (SoC), there are two 32-bit registers for enabling interrupts of peripheral devices. These registers, along with other interrupt control registers, reside in IO memory at `INT_REGS_BASE` which is defined in *i*nclude/traps.h. Unfortunately, a lot of details on these registers are probably intentionally made unclear in *B*CM2835 ARM Peripherals.

Now we have understood the initialization of the mini UART. After initialization, the mini UART is ready to send data to and receive data from the serial port via pins 14 and 15. Once a byte is received, the mini UART will generate an interrupt for the driver to handle. Below is our interrupt handler invoked in `trap()`.

```
miniuartintr(void)
{
  consoleintr(uartgetc);
}
```

In the interrupt handler, the console's interrupt handler `consoleintr()` is simply called with the function `uartgetc()` passed to `consoleintr()`. This is a simple method for handling interrupts in a small OS like xv6 as only the console uses the UART and the console's handling time is short. Another way is, like Linux, to use `uartgetc()` to get the bytes directly and repeatedly until no more in the UART FIFO buffer, and then store the bytes in the kernel buffer, but schedule the console handler as bottomhalf or tasklet. After all other interrupts are handled, run the bottomhalves and tasklets. We will look at the console code in detail later.

Basically our UART driver only provide two functions for high-level kernel users to invoke, `uartgetc()` and `uartputc()`. Below is the code for `uartgetc()`.

```
static int
uartgetc(void)
{
        if(inw(AUX_MU_LSR_REG)&0x1) return inw(AUX_MU_IO_REG);
        else return -1;
}
```

`AUX_MU_LSR_REG` is a line status register that shows the data status of the UART. If its least significant bit is set, it means that data is ready to read. If its bit 5 is set, it means the transmit FIFO can accept at least one byte. `inw()` and `outw()` are for reading and writing IO memory or registers. Basically, in `uartgetc()`, if the data is ready, it returns a byte read from the UART FIFO buffer via the IO data register `AUX_MU_IO_REG`; otherwise it returns -1.

Now let us look at `uartputc()`.

```
void uartputc(uint c)
{
        if(c=='\n') {
                while(1) if(inw(AUX_MU_LSR_REG) & 0x20) break;
                outw(AUX_MU_IO_REG, 0x0d); // add CR before LF
        }
        while(1) if(inw(AUX_MU_LSR_REG) & 0x20) break;
        outw(AUX_MU_IO_REG, c);
}
```

In `uartputc()`, it checks bit 5 of `AUX_MU_LSR_REG` via `inw(AUX_MU_LSR_REG) & 0x20`. If it is true, the UART is ready to accept at least one byte and the byte is sent using `outw(AUX_MU_IO_REG, c)` via `AUX_MU_IO_REG`; otherwise, it keeps busy waiting until the condition is true. The only additional work is when the character is `'\n'` we need to send an extra linefeed 0xd to make it compatible with old console format.

Now we have got a better idea of a slightly more complex device driver due to the complex hardware involved for UART. We will look at a simpler device driver for RAMDISK that has a very simple device, the memory.

RAMDISK

The RAMDISK consists of a number of 512-byte sectors, which are contiguously stored in a memory space. Each sector has a unique number and is easily addressable with the number.

The RAMDISK driver code is in source/memide.c. It handles read and write requests from the high-level kernel user, the file system, to the RAMDISK via the following buffer data structure.

```
struct buf {
  int flags;
  uint dev;
  uint sector;
  struct buf *prev; // LRU cache list
  struct buf *next;
  struct buf *qnext; // disk queue
  uchar data[512];
};
```

The `buf` structure contains the status information and data of a sector in the RAMDISK. `flags` tells if the sector is dirty or valid. If a sector is dirty, it means the sector has been modified in the buffer and should be written into the RAMDISK to update the sector accordingly. If a sector is not valid, it means the sector should be read from the RAMDISK to update the memory buffer. The reason to use memory buffers is to improve disk performance, as disk read/write is very slow compared with memory read/write. For a sector that is repeatedly read/write, we keep it in memory buffer to avoid slow disk operations. However, in the case of RAMDISK, there is no performance advantage as both the sector and buffer are in memory. However, since using memory buffers is a standard for all disk devices, the RAMDISK driver has to use the memory buffers due to consistent interface to the high-level file system.

The `buf` structure also contains other fields such as device number `dev` and pointers for various linked lists, the use of which will be described in Chapter 7.

The RAMDISK is initialized by the following `ideinit()` function.

```
void ideinit(void)
{
```

```
  memdisk = _binary_fs_img_start;
  disksize = div(((uint)_binary_fs_img_end - (uint)_binary_fs_img_start), 512);
}
```

In `ideinit()`, the starting address of the disk image in memory is put into `memdisk`. Then `disksize`, the number of sectors, is calculated. The disk image was pre-made by *u*prog/mkfs and is a file system that contains all the user-space executable programs and a README file. The file system follows a simple Unix file system with data blocks (sectors) connected by index nodes (inodes). More details on the file system will be in Chapter 7.

Since it is a memory device, there is no interrupt to handle for the RAMDISK. So the interrupt handler `ideintr()` is empty.

The function of the RAMDISK driver is to read and write disk sectors using the `buf` structure. So the following `iderw()` is the key function in the driver.

```
void iderw(struct buf *b)
{
  uchar *p;

  if(!(b->flags & B_BUSY))
    panic("iderw: buf not busy");
  if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
    panic("iderw: nothing to do");
  if(b->dev != 1)
    panic("iderw: request not for disk 1");
  if(b->sector >= disksize)
    panic("iderw: sector out of range");

  p = memdisk + b->sector*512;

  if(b->flags & B_DIRTY){
    b->flags &= ~B_DIRTY;
    memmove(p, b->data, 512);
  } else
    memmove(b->data, p, 512);
  b->flags |= B_VALID;
}
```

Since RAMDISK has no complex hardware, read/write operations are very simple in `iderw()`, just memory copy with `memmove()`. The basic idea of `iderw()` is also simple: if the block in the `buf` structure is dirty, write the block into the RAMDISK; if the block is not valid, read the block from the RAMDISK.

The first part of the code is sanity check, making sure there is no error in the kernel code. Normally when `iderw()` is invoked, the `flags` of the block (or sector) should be set busy to avoid data race, and it is either dirty or not valid otherwise there is no need to read or write. Also the device number `dev` has to be correct and the sector number needs to be valid.

After the sanity check, `p = memdisk + b->sector*512;` finds the memory address of the sector in the RAMDISK. If the block of the memory buffer is dirty, write the block into the corresponding sector in the RAMDISK and clear the dirty flag of the block.

Otherwise, the block must be not valid, and `iderw()` reads the sector in the RAMDISK and copies it to the block in the memory buffer and sets the valid flag of the block.

The operations of this RAMDISK driver are at the lowest layer of the file system of xv6. We will see other layers in the file system in Chpater 7.

Chapter 6

Scheduler

Chapter 7

File System

Chapter 8

Concurrency