Created: June 18 2018 10:58:00 Last Updated: Jun 20 2018 01:17:37

Lab 5 Report

Shuyang Cao

Jun 20 2018

The File System

In this lab, we won't implement the entire file syetem, but implement only certain key components. In particular, we implement reading blocks into the block cache and flushing them back to disk; allocating disk blocks; mapping file offsets to disk blocks; and the reading, writing, opening IPC interfaces.

Disk Access

Instead of taking the conventional "monolithic" operating system strategy of adding an IDE disk driver to the kernel along with the necessary system calls to allow the file system to access it, we instead implement the IDE disk driver as part of the user-level file system enviroment. To do so, we need to give the privileges the file system enviroment needs to allow it to implement disk access itself. As long as we rely on polling, "programmed I/O" (PIO)-based disk access and do not use disk interrupts, it's easy to implement disk access in user space.

The x86 processor uses the IOPL bits in the EFLAGS register to determine whether protected-mode code is allowed to perform special device I/O instructions such as the IN and OUT instructions. Only processes with CS PL lower than their own IOPL are allowed to perform special device I/O. Since all of the IDE disk registers we need to access are located in the x86's I/O space rather than being memory-mapped, giving "I/O privilege" to the file system environment is the only thing we need to do in order to allow the file system to access these registers. In effect, the IOPL bits in the EFLAGS register provides the kernel with a simple "all-or-nothing" method of controlling whether user-mode code can access I/O space. In JOS, we want the file system environment to be able to access I/O space, but we do not want any other environments to be able to access I/O space at all.

Exercise 1

> `i386_init` identifies the file system environment by passing the type `ENV_TYPE_FS` to your environment creation function, `env_create`. Modify `env_create` in `env.c`, so that it gives the file system environment I/O privilege, but never gives that privilege to any other environment.

> Make sure you can start the file environment without causing a General Protection fault. You should pass the “fs i/o” test in `make grade` .

As mentioned in last section, we need to set IOPL larger than CSPL for FS enviroment while setting IOPL to zero for other enviroments.

The code for this exercise is shown below.

```
diff --git a/kern/env.c b/kern/env.c
index 3ddc120..6533465 100644
--- a/kern/env.c
+++ b/kern/env.c
@@ -424,6 +424,9 @@ env_create(uint8_t *binary, enum EnvType type)
        if ((err_code = env_alloc(&e, 0)) != 0)
                panic("env_alloc: %e", err_code);

+       if (type == ENV_TYPE_FS)
+               e->env_tf.tf_eflags |= FL_IOPL_3;
+
        load_icode(e, binary);
        e->env_type = type;
 }
```

Question 1

> Do you have to do anything else to ensure that this I/O privilege setting is saved and restored properly when you subsequently switch from one environment to another? Why?

The Block Cache

In our file system, we will implement a simple “buffer cache”, essentially a block cache, with the help of the processor’s virtual memroy system. The JOS file system is limited to handling disks of size 3GB or less. We reserve a large, fixed 3GB region of the file system environment’s address space, from 0x10000000 (`DISKMAP`) up to 0xD0000000 (`DISKMAP` + `DISKMAX`), as a “memory mapped” version of the disk.

Since our file system environment has its own virtual address space independent of the virtual address spaces of all other environments in the system, and the only thing the file system environment needs to do is to implement file access, it is reasonable to reserve most of the file system environment’s address space in this way. Of course, it would take a long time to read the entire disk into memory, so instead we’ll implement a form of demand paging, wherein we only allocate pages in the disk map region and read the corresponding block from the disk in response to a page fault in this region. This way, we can pretend that the entire disk is in memory.

Exercise 2

> Implement the `bc_pgfault` and `flush_block` functions in `fs/bc.c` . `bc_pgfault` is a page fault handler, just like the one we wrote in the previous lab for copy-on-write fork, except that its job is to load pages in from the disk in response to a page fault. When writing this, keep in mind that
>
> > (1) addr may not be aligned to a block boundary
> > (2) ide_read operates in sectors, not blocks.
>
> The `flush_block` function should write a block out to disk if necessary. `flush_block` shouldn’t do anything if the block isn’t even in the block cache (that is, the page isn’t mapped) or if it’s not dirty. We will use the VM hardware to keep track of whether a disk block has been modified since it was last read from or written to disk. To see whether a block needs writing, we can just look to see if the `PTE_D` “dirty” bit is set in the `uvpt` entry. After writing the block to disk, `flush_block` should clear the `PTE_D` bit using `sys_page_map` .
>
> Use `make grade` to test your code. Your code should pass “check_bc”, “check_super”, and “check_bitmap”.

Remember to round down the address passed to `flush_block`

The code for this exercise is shown below.

```
diff --git a/fs/bc.c b/fs/bc.c
index e3922c4..86b4f86 100644
--- a/fs/bc.c
+++ b/fs/bc.c
@@ -48,6 +48,12 @@ bc_pgfault(struct UTrapframe *utf)
        // the disk.
        //
        // LAB 5: you code here:
+       addr = ROUNDDOWN(addr, PGSIZE);
+       if ((r = sys_page_alloc(0, addr, PTE_U | PTE_P | PTE_W)) < 0)
+               panic("bc_pgfault, sys_page_alloc: %e\n", r);
+
+       if ((r = ide_read(blockno * BLKSECTS, addr, BLKSECTS)) < 0)
+               panic("bc_pgfault, ide_read: %e\n", r);

        // Clear the dirty bit for the disk block page since we just read the
        // block from disk
@@ -77,7 +83,17 @@ flush_block(void *addr)
                panic("flush_block of bad va %08x", addr);

        // LAB 5: Your code here.
-       panic("flush_block not implemented");
+       // panic("flush_block not implemented");
+       int r;
+
+       addr = ROUNDDOWN(addr, PGSIZE);
+       if (va_is_mapped(addr) && va_is_dirty(addr)) {
+               if ((r = ide_write(blockno * BLKSECTS, addr, BLKSECTS)) < 0)
+                       panic("flush_block, ide_write: %e\n", r);
+
+               if ((r = sys_page_map(0, addr, 0, addr, uvpt[PGNUM(addr)] & PTE_SYSCALL)) < 0)
+                       panic("flush_block, sys_page_map: %e\n", r);
+       }
 }

 // Test that the block cache works, by smashing the superblock and
```

The Block Bitmap & Exericse 3

After `fs_init` sets the bitmap pointer, we can treat bitmap as a packed array of bits, one for each block on the disk.

> Use `free_block` as a model to implement `alloc_block` in `fs/fs.c`, which should find a free disk block in the bitmap, mark it used, and return the number of that block. When you allocate a block, you should immediately flush the changed bitmap block to disk with flush_block, to help file system consistency.
>
> Use `make grade` to test your code. Your code should now pass “alloc_block”.

Just loop through the bitmap to find a free block and follow the hint.

The code for this exercise is shown below.

```
diff --git a/fs/fs.c b/fs/fs.c
index 45ecaf8..261151c 100644
--- a/fs/fs.c
+++ b/fs/fs.c
@@ -62,7 +62,15 @@ alloc_block(void)
        // super->s_nblocks blocks in the disk altogether.

        // LAB 5: Your code here.
-       panic("alloc_block not implemented");
+       //panic("alloc_block not implemented");
+       for (uint32_t blockno = 0; blockno < super->s_nblocks; ++blockno) {
+               if (block_is_free(blockno)) {
+                       bitmap[blockno/32] &= ~(1 << (blockno % 32));
+                       flush_block(bitmap);
+                       return blockno;
+               }
+       }
+
        return -E_NO_DISK;
 }
```

File Operations & Exercise 4

> Implement `file_block_walk` and `file_get_block`. `file_block_walk` maps from a block offset within a file to the pointer for that block in the struct File or the indirect block, very much like `what pgdir_walk` did for page tables. `file_get_block` goes one step further and maps to the actual disk block, allocating a new one if necessary.
>
> Use `make grade` to test your code. Your code should pass “file_open”, “file_get_block”, and “file_flush/file_truncated/file rewrite”, and “testfile”.

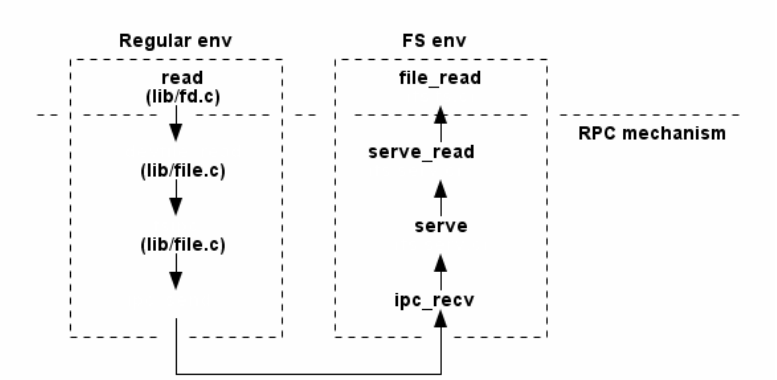
Review `pgdir_walk` and follow the hint. It’s easy to finish this part. The code for this exercise is shown below.

```
diff --git a/fs/fs.c b/fs/fs.c
index 261151c..3454db4 100644
--- a/fs/fs.c
+++ b/fs/fs.c
@@ -142,8 +142,35 @@ fs_init(void)
 static int
 file_block_walk(struct File *f, uint32_t filebno, uint32_t **ppdiskbno, bool alloc)
 {
-	// LAB 5: Your code here.
-	panic("file_block_walk not implemented");
+    // LAB 5: Your code here.
+	// panic("file_block_walk not implemented");
+	int r;
+
+	if (filebno >= NDIRECT + NINDIRECT)
+		return -E_INVAL;
+
+	if (filebno < NDIRECT) {
+		if (ppdiskbno)
+			*ppdiskbno = f->f_direct + filebno;
+		return 0;
+	}
+
+	if (!f->f_indirect) {
+		if (!alloc)
+			return -E_NOT_FOUND;
+
+		if ((r = alloc_block()) < 0)
+			return -E_NO_DISK;
+
+		f->f_indirect = r;
+		memset(diskaddr(r), 0, BLKSIZE);
+		flush_block(diskaddr(r));
+	}
+
+	if (ppdiskbno)
+		*ppdiskbno = (uint32_t *)diskaddr(f->f_indirect) + filebno - NDIRECT;
+
+    return 0;
 }

 // Set *blk to the address in memory where the filebno'th
@@ -157,8 +184,25 @@ file_block_walk(struct File *f, uint32_t filebno, uint32_t **ppdiskbno, bool all
 int
 file_get_block(struct File *f, uint32_t filebno, char **blk)
 {
-	// LAB 5: Your code here.
-	panic("file_get_block not implemented");
+	// LAB 5: Your code here.
+	// panic("file_get_block not implemented");
+	int r;
+	uint32_t *ppdiskbno;
+
+	if ((r = file_block_walk(f, filebno, &ppdiskbno, true)) < 0)
+		return r;
+
+	if (*ppdiskbno == 0) {
+		if ((r = alloc_block()) < 0)
+			return -E_NO_DISK;
+
+		*ppdiskbno = r;
+		memset(diskaddr(r), 0, BLKSIZE);
+		flush_block(diskaddr(r));
+	}
+
+	*blk = diskaddr(*ppdiskbno);
+	return 0;
 }

 // Try to find a file named "name" in dir.  If so, set *file to it.
diff --git a/fs/serv.c b/fs/serv.c
index 76c1d99..07c6d00 100644
--- a/fs/serv.c
+++ b/fs/serv.c
@@ -214,7 +214,19 @@ serve_read(envid_t envid, union Fsipc *ipc)
 		cprintf("serve_read %08x %08x %08x\n", envid, req->req_fileid, req->req_n);
```

```
        // Lab 5: Your code here:
-       return 0;
+       struct OpenFile *o;
+       int r;
+
+       if ((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
+               return r;
+
+       size_t req_n = req->req_n > PGSIZE ? PGSIZE : req->req_n;
+
+       if ((r = file_read(o->o_file, ret->ret_buf, req_n, o->o_fd->fd_offset)) < 0)
+               return r;
+
+       o->o_fd->fd_offset += r;
+       return r;
 }
```

The file system interface

Now that we have the necessary functionality within the file system environment itself, we must make it accessible to other environments that wish to use the file system. Since other environments can't directly call functions in the file system environment, we'll expose access to the file system environment via a remote procedure call, or RPC, abstraction, built atop JOS's IPC mechanism. Graphically, here's what a call to the file system server (say, read) looks like



Recall that JOS's IPC mechanism lets an environment send a single 32-bit number and, optionally, share a page. To send a request from the client to the server, we use the 32-bit number for the request type (the file system server RPCs are numbered, just like how syscalls were numbered) and store the arguments to the request in a union `Fsipc` on the page shared via the IPC. On the client side, we always share the page at `fsipcbuf` ; on the server side, we map the incoming request page at `fsreq` (0x0ffff000).

The server also sends the response back via IPC. We use the 32-bit number for the function's return code. For most RPCs, this is all they return. `FSREQ_READ` and `FSREQ_STAT` also return data, which they simply write to the page that the client sent its request on. There's no need to send this page in the response IPC, since the client shared it with the file system server in the first place. Also, in its response, `FSREQ_OPEN` shares with the client a new "Fd page".

Exercise 5

> Implement `serve_read` in `fs/serv.c` .
>
> `serve_read` 's heavy lifting will be done by the already-implemented `file_read` in `fs/fs.c` (which, in turn, is just a bunch of calls to `file_get_block`). `serve_read` just has to provide the RPC interface for file reading. Look at the comments and code in `serve_set_size` to get a general idea of how the server functions should be structured.
>
> Use `make grade` to test your code. Your code should pass "serve_open/file_stat/file_close" and "file_read" for a score of 70/150.

By imitating `serve_set_size` , it's easy to implement `serve_read` . The code for this exercise is shown below.

```
diff --git a/fs/serv.c b/fs/serv.c
index 76c1d99..07c6d00 100644
--- a/fs/serv.c
+++ b/fs/serv.c
@@ -214,7 +214,19 @@ serve_read(envid_t envid, union Fsipc *ipc)
                cprintf("serve_read %08x %08x %08x\n", envid, req->req_fileid, req->req_n);

        // Lab 5: Your code here:
-       return 0;
+       struct OpenFile *o;
+       int r;
+
+       if ((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
+               return r;
+
+       size_t req_n = req->req_n > PGSIZE ? PGSIZE : req->req_n;
+
+       if ((r = file_read(o->o_file, ret->ret_buf, req_n, o->o_fd->fd_offset)) < 0)
+               return r;
+
+       o->o_fd->fd_offset += r;
+       return r;
 }
```

Exercise 6

> Implement `serve_write` in `fs/serv.c` and `devfile_write` in `lib/file.c`.
>
> Use `make grade` to test your code. Your code should pass “file_write”, “file_read after file_write”, “open”, and “large file” for a score of 90/150.

The implementation for `serve_write` is similar to `serve_read`. The majority of `devfile_write`'s work is invoke a RPC. The code for this exercise is shown below.

```
diff --git a/fs/serv.c b/fs/serv.c
index 07c6d00..deda7ea 100644
--- a/fs/serv.c
+++ b/fs/serv.c
@@ -241,7 +241,21 @@ serve_write(envid_t envid, struct Fsreq_write *req)
                cprintf("serve_write %08x %08x %08x\n", envid, req->req_fileid, req->req_n);

        // LAB 5: Your code here.
-       panic("serve_write not implemented");
+       // panic("serve_write not implemented");
+       struct OpenFile *o;
+       int r;
+
+       if ((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
+               return r;
+
+       size_t req_n = req->req_n > PGSIZE ? PGSIZE : req->req_n;
+
+       // file_write will extend the file automatically if necessary.
+       if ((r = file_write(o->o_file, req->req_buf, req_n, o->o_fd->fd_offset)) < 0)
+               return r;
+
+       o->o_fd->fd_offset += r;
+       return r;
 }

 // Stat ipc->stat.req_fileid.  Return the file's struct Stat to the
diff --git a/lib/file.c b/lib/file.c
index 39025b2..f642fda 100644
--- a/lib/file.c
+++ b/lib/file.c
@@ -141,7 +141,21 @@ devfile_write(struct Fd *fd, const void *buf, size_t n)
        // remember that write is always allowed to write *fewer*
        // bytes than requested.
        // LAB 5: Your code here
-       panic("devfile_write not implemented");
+       // panic("devfile_write not implemented");
+       int r;
+
+       if (n > sizeof(fsipcbuf.write.req_buf))
+               n = sizeof(fsipcbuf.write.req_buf);
+
+       fsipcbuf.write.req_fileid = fd->fd_file.id;
+       fsipcbuf.write.req_n = n;
+       memcpy(fsipcbuf.write.req_buf, buf, n);
+
+       if ((r = fsipc(FSREQ_WRITE, NULL)) < 0)
+               return r;
+
+       assert(r <= n);
+       return r;
 }

 static int
```

Spawning Processes & Exercise 7

`lib/spawn.c` provides `spawning`, which creates a new environment, loads a program image from the file system into it, and then starts the child environment running this program. JOS implemented `spawn` rather than a UNIX-style `exec` because spawn is easier to implement from user space in “exokernel fashion”, without special help from the kernel.

> `spawn` relies on the new syscall `sys_env_set_trapframe` to initialize the state of the newly created environment. Implement `sys_env_set_trapframe` in `kern/syscall.c` (don’t forget to dispatch the new system call in `syscall()`).
>
> Test your code by running the user/spawnhello program from `kern/init.c`, which will attempt to spawn /hello from the file system.
>
> Use `make grade` to test your code.

Remember to check the sanity of the address passed to JOS kernel. Besides, don’t forget to set IOPL to a appropriate value. Zero is a good choice. The code for this exercise is shown below.

```
diff --git a/kern/syscall.c b/kern/syscall.c
index 39e39ad..b7243d0 100644
--- a/kern/syscall.c
+++ b/kern/syscall.c
@@ -141,7 +141,21 @@ sys_env_set_trapframe(envid_t envid, struct Trapframe *tf)
        // LAB 5: Your code here.
        // Remember to check whether the user has supplied us with a good
        // address!
-       panic("sys_env_set_trapframe not implemented");
+       // panic("sys_env_set_trapframe not implemented");
+       struct Env *e;
+       int r;
+
+       if ((r = envid2env(envid, &e, true)) < 0)
+               return -E_BAD_ENV;
+
+       user_mem_assert(curenv, tf, sizeof(struct Trapframe), PTE_U | PTE_P);
+
+       e->env_tf = *tf;
+       e->env_tf.tf_cs |= 3;
+       e->env_tf.tf_eflags |= FL_IF;
+       e->env_tf.tf_eflags &= ~FL_IOPL_MASK;
+
+       return 0;
 }

 // Set the page fault upcall for 'envid' by modifying the corresponding struct
@@ -422,6 +436,7 @@ syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4,
                case SYS_exofork:       return sys_exofork();
                case SYS_env_set_status:        return sys_env_set_status((envid_t)a1, (int)a2);
+               case SYS_env_set_trapframe:     return sys_env_set_trapframe((envid_t)a1, (struct Trapframe *)a2);
                case SYS_env_set_pgfault_upcall: return sys_env_set_pgfault_upcall((envid_t)a1, (void *)a2);
                case SYS_yield: sys_yield(); return 0;
                case SYS_ipc_try_send: return sys_ipc_try_send((envid_t)a1, (uint32_t)a2, (void *)a3, (unsigned int)a4);
```

Sharing library state across fork and spawn

The UNIX file descriptors are a general notion that also encompasses pipes, console I/O, etc. In JOS, each of these device types has a corresponding `struct Dev`, with pointers to the functions that implement read/write/etc. for that device type. `lib/fd.c` implements the general UNIX-like file descriptor interface on top of this. Each struct Fd indicates its device type, and most of the functions in `lib/fd.c` simply dispatch operations to functions in the appropriate `struct Dev`.

`lib/fd.c` also maintains the file descriptor table region in each application environment's address space, starting at `FDTABLE`. This area reserves a page's worth (4KB) of address space for each of the up to `MAXFD` (currently 32) file descriptors the application can have open at once. At any given time, a particular file descriptor table page is mapped if and only if the corresponding file descriptor is in use. Each file descriptor also has an optional "data page" in the region starting at FILEDATA, which devices can use if they choose.

JOS chose to share file descriptor state across fork and spawn, but file descriptor state is kept in user-space memory. On fork, the memory will be marked copy-on-write, so the state will be duplicated rather than shared. On spawn, the memory will be left behind, not copied at all.

We will set an otherwise-unused bit (PTE_SHARE) in the page table entries to notify `fork` certain regions of memory are used by the "library operating system" and should always be shared.

Exercise 8

> Change `duppage` in `lib/fork.c` to follow the new convention. If the page table entry has the PTE_SHARE bit set, just copy the mapping directly.
>
> Likewise, implement copy_shared_pages in `lib/spawn.c`. It should loop through all page table entries in the current process (just like fork did), copying any page mappings that have the PTE_SHARE bit set into the child process.
>
> Use `make run-testpteshare` to check that your code is behaving properly. You should see lines that say "fork handles PTE_SHARE right" and "spawn handles PTE_SHARE right".
>
> Use `make run-testfdsharing` to check that file descriptors are shared properly. You should see lines that say "read in child succeeded" and "read in parent succeeded".

As long as we understand what is a share page in last section, the logic for the change is not complicate. The code for this exercise is shown below.

```
diff --git a/lib/fork.c b/lib/fork.c
index 2e54f5f..0151203 100644
--- a/lib/fork.c
+++ b/lib/fork.c
@@ -67,21 +67,21 @@ duppage(envid_t envid, unsigned pn)

        // LAB 4: Your code here.
        // panic("duppage not implemented");
-       void *addr = (void *)(pn*PGSIZE);
-       int perm = PGOFF(uvpt[pn]) & PTE_SYSCALL;
+       void *addr = (void *)(pn * PGSIZE);
+       int perm = uvpt[pn] & PTE_SYSCALL;

-       if (perm & (PTE_W | PTE_COW)) {
+       if ((perm & PTE_SHARE) || !(perm & (PTE_W | PTE_COW))) {
+               if ((r = sys_page_map(0, addr, envid, addr, perm)) < 0)
+                       return r;
+       } else {
                perm |= PTE_COW;
                perm &= ~PTE_W;

                if ((r = sys_page_map(0, addr, envid, addr, perm)) < 0)
-                       panic("Failed to duppage on child enviroment: %e\n", r);
+                       return r;

                if ((r = sys_page_map(0, addr, 0, addr, perm)) < 0)
-                       panic("Failed to duppage on self enviroment: %e\n", r);
-       } else {
-               if ((r = sys_page_map(0, addr, envid, addr, perm)) < 0)
-                       panic("Failed to duppage on child enviroment: %e\n", r);
+                       return r;
        }

        return 0;
diff --git a/lib/spawn.c b/lib/spawn.c
index 9d0eb07..c11ef20 100644
--- a/lib/spawn.c
+++ b/lib/spawn.c
@@ -302,6 +302,26 @@ static int
 copy_shared_pages(envid_t child)
 {
        // LAB 5: Your code here.
+       int r;
+
+       for (size_t i = 0; i < NPDENTRIES; ++i) {
+               if (uvpd[i] & PTE_P) {
+                       for (size_t j = 0; j < NPTENTRIES; ++j) {
+                               size_t pn = (i << (PDXSHIFT - PTXSHIFT)) + j;
+
+                               if (pn == PGNUM(USTACKTOP))
+                                       return 0;
+
+                               int perm = uvpt[pn];
+                               if ((perm & PTE_P) && (perm & PTE_SHARE)) {
+                                       void *addr = (void *)PGADDR(i, j, 0);
+                                       if ((r = sys_page_map(0, addr, child, addr, perm & PTE_SYSCALL)) < 0)
+                                               return r;
+                               }
+                       }
+               }
+       }
+
        return 0;
 }
```

The keyboard interface & Exercise 9

For the shell to work, we need a way to type at it. QEMU has been displaying output we write to the CGA display and the serial port, but so far we've only taken input while in the kernel monitor. In QEMU, input typed in the graphical window appear as input from the keyboard to JOS, while input typed to the console appear as characters on the serial port. `kern/console.c` already contains the keyboard and serial drivers that have been used by the kernel monitor since lab 1, but now we need to attach these to the rest of the system.

> In your `kern/trap.c`, call `kbd_intr` to handle trap `IRQ_OFFSET+IRQ_KBD` and `serial_intr` to handle trap `IRQ_OFFSET+IRQ_SERIAL`.

All we need to do is adding another two trap handlers. The code for this exercise is shown below.

```
diff --git a/kern/trap.c b/kern/trap.c
index 8e48f15..073be99 100644
--- a/kern/trap.c
+++ b/kern/trap.c
@@ -220,6 +220,10 @@ trap_dispatch(struct Trapframe *tf)
                case IRQ_OFFSET + IRQ_TIMER: lapic_eoi();
                                                                        sched_yield();
                                                                        return;
+               case IRQ_OFFSET + IRQ_KBD:      kbd_intr();
+                                                                      return;
+               case IRQ_OFFSET + IRQ_SERIAL:   serial_intr();
+                                                                             return;
                case T_SYSCALL: tf->tf_regs.reg_eax = syscall(
                                                                      tf->tf_regs.reg_eax,
                                                                      tf->tf_regs.reg_edx,
```

The Shell & Exercise 10

Note that the user library routine `cprintf` prints straight to the console, without using the file descriptor code. This is great for debugging but not great for piping into other programs. To print output to a particular file descriptor (for example, 1, standard output), use `fprintf(1, "...", ...)`. `printf("...", ...)` is a short-cut for printing to FD 1.

> The shell doesn't support I/O redirection. It would be nice to run `sh < script` instead of having to type in all the commands in the script by hand. Add I/O redirection for `<` to `user/sh.c`.
>
> Test your implementation by typing `sh < script` into your shell.
>
> Run `make run-testshell` to test your shell. testshell simply feeds commands in `fs/testshell.sh` into the shell and then checks that the output matches `fs/testshell.key`.

The input redirection is just a mirror of the output redirection, which has been implemented in `user/sh.c`. The code for this exercise is shown below.

```
diff --git a/kern/pmap.c b/kern/pmap.c
index 047f266..465475e 100644
--- a/kern/pmap.c
+++ b/kern/pmap.c
@@ -637,7 +637,7 @@ page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)

        pte_t *pt_entry = pgdir_walk(pgdir, va, false);

-       if (pt_entry == NULL)
+       if (pt_entry == NULL || !(*pt_entry & PTE_P))
                return NULL;

        if (pte_store)  // if pte_store is not zero
diff --git a/user/sh.c b/user/sh.c
index 26f501a..464cdcb 100644
--- a/user/sh.c
+++ b/user/sh.c
@@ -55,7 +55,15 @@ again:
                        // then close the original 'fd'.

                        // LAB 5: Your code here.
-                       panic("< redirection not implemented");
+                       if ((fd = open(t, O_RDONLY)) < 0) {
+                               cprintf("open %s for write: %e\n", t, fd);
+                               exit();
+                       }
+                       if (fd != 0) {
+                               dup(fd, 0);
+                               close(fd);
+                       }
+                       // panic("< redirection not implemented");
                        break;

                case '>':       // Output redirection
```

Challenge

> The block cache has no eviction policy. Once a block gets faulted in to it, it never gets removed and will remain in memory forevermore. Add eviction to the buffer cache. Using the PTE_A “accessed” bits in the page tables, which the hardware sets on any access to a page, you can track approximate usage of disk blocks without the need to modify every place in the code that accesses the disk map region. Be careful with dirty blocks.

Instead of cleanning up pages at a specific frequency, we add a counter to `bc_pgfault` , in which `evict_page` is called every `EVIC_INTER` bc page faults.

`evict_page` executes the following eviction policy:

1. Pages not accessed are unmapped.
2. Access bits on pages accessed are unset.
3. Dirty pages are flushed out and dirty bits are unset.

The code for this challenge is shown below.

```
diff --git a/fs/bc.c b/fs/bc.c
index 86b4f86..9939a39 100644
--- a/fs/bc.c
+++ b/fs/bc.c
@@ -29,10 +29,16 @@ va_is_dirty(void *va)
 static void
 bc_pgfault(struct UTrapframe *utf)
 {
+       static uint32_t pgfault_count = 0;
        void *addr = (void *) utf->utf_fault_va;
        uint32_t blockno = ((uint32_t)addr - DISKMAP) / BLKSIZE;
        int r;

+       if (++pgfault_count > EVIC_INTER) {
+               evict_page();
+               pgfault_count = 0;
+       }
+
        // Check that the fault was within the block cache region
        if (addr < (void*)DISKMAP || addr >= (void*)(DISKMAP + DISKSIZE))
                panic("page fault in FS: eip %08x, va %08x, err %04x",
@@ -96,6 +102,40 @@ flush_block(void *addr)
        }
 }

+
+// clean up block cache every 100 bc page faults.
+// strategy:
+//             1. unmap page not accessed
+//             2. unset access bit on pages accessed
+//             3. flush out dirty pages and unset their accessed bit
+void
+evict_page()
+{
+       // this code is buggy if DISKMAP is not page directory aligned (NPTEENTRIES * PGSIZE);
+       static_assert(DISKMAP % PTSIZE == 0);
+
+       for (size_t i = PDX(DISKMAP); i < PDX(ROUNDUP(DISKMAP + DISKSIZE, PTSIZE)); ++i) {
+               if (uvpd[i] & PTE_P) {
+                       pte_t *pgtable = (pte_t *)(uvpt + i * NPTENTRIES);
+
+                       for (size_t j = 0; j < NPTENTRIES; ++j) {
+                               if (pgtable[j] & PTE_P) {
+                                       void *addr = PGADDR(i, j, 0);
+
+                                       if (pgtable[j] & PTE_A) {
+                                               if (pgtable[j] & PTE_D)
+                                                       flush_block(addr);
+
+                                               sys_page_map(0, addr, 0, addr, pgtable[j] & PTE_SYSCALL);
+                                       } else {
+                                               sys_page_unmap(0, addr);
+                                       }
+                               }
+                       }
+               }
+       }
+}
+
 // Test that the block cache works, by smashing the superblock and
 // reading it back.
 static void
diff --git a/fs/fs.h b/fs/fs.h
index 0350d78..cf84331 100644
--- a/fs/fs.h
+++ b/fs/fs.h
@@ -11,6 +11,9 @@
 /* Maximum disk size we can handle (3GB) */
 #define DISKSIZE       0xC0000000

+/* Block cache eviction interval */
+#define EVIC_INTER  100
+
 struct Super *super;           // superblock
 uint32_t *bitmap;              // bitmap blocks mapped in memory

@@ -26,6 +29,7 @@ void* diskaddr(uint32_t blockno);
```

```
 bool   va_is_mapped(void *va);
 bool   va_is_dirty(void *va);
 void   flush_block(void *addr);
+void    evict_page();
 void   bc_init(void);

 /* fs.c */
```

Grade

Finally, we got our grade.

```
internal FS tests [fs/test.c]: OK (30.5s)
  fs i/o: OK
  check_bc: OK
  check_super: OK
  check_bitmap: OK
  alloc_block: OK
  file_open: OK
  file_get_block: OK
  file_flush/file_truncate/file rewrite: OK
testfile: OK (1.9s)
  serve_open/file_stat/file_close: OK
  file_read: OK
  file_write: OK
  file_read after file_write: OK
  open: OK
  large file: OK
spawn via spawnhello: OK (1.2s)
Protection I/O space: OK (1.6s)
PTE_SHARE [testpteshare]: OK (1.5s)
PTE_SHARE [testfdsharing]: OK (1.8s)
start the shell [icode]: Timeout! OK (30.6s)
testshell: OK (2.9s)
primespipe: OK (8.2s)
Score: 150/150
```

End of Lab 5 Report

Email: caoshuyang@pku.edu.cn GitHub: JOS-Lab