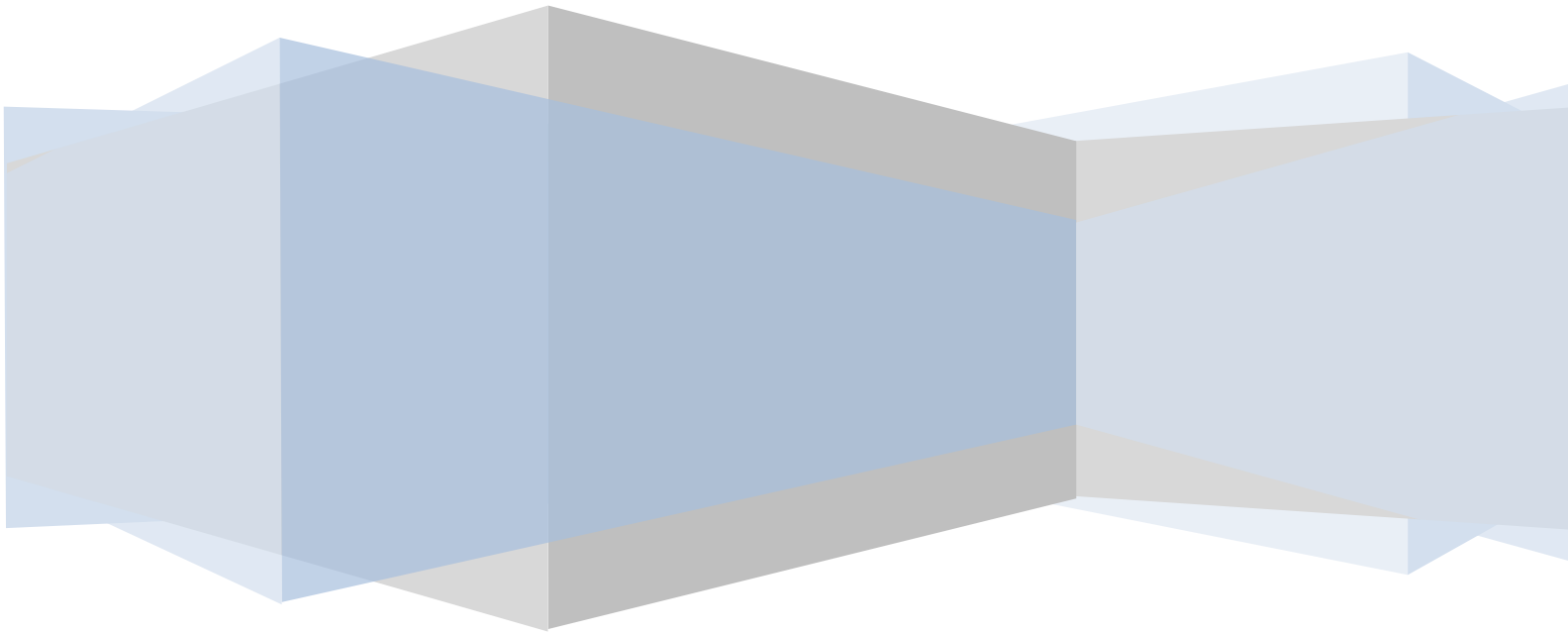
ANALISIS DEL SISTEMA OPERATIVO XV6

Realizado por: Juan Carlos López Muñoz



ÍNDICE

1. INTRODUCCIÓN Al SISTEMA OPERATIVO XV6 3
2. EXPLICACION DEL MANEJO PARA EJECUTAR XV6 3
3. OBJETIVO DEL PROYECTO 3
3.1 Clasificación de los componentes básicos del SO 4
3.1.1 Clasificación sobre el Gestión de procesos. 4
3.1.2. Clasificación sobre el Administración de memoria principal 30
3.1.3 Clasificación sobre el Gestión de E/S 39
3.1.4 Clasificación sobre el Administración de ficheros 45
4. CONCLUSIÓN 60

1. INTRODUCCIÓN Al SISTEMA OPERATIVO XV6

XV6 es un sistema operativo creado por el MIT (Massachusetts Institute of Technology) con motivos educacionales y desarrollados en el 2006. Este sistema operativo está basado en la versión 6 de UNIX y es una reimplementación de éste para la arquitectura de procesadores intel x86. Se trata de un sistema operativo monolítico escrito en el lenguaje de programación C. Además su interfaz de usuario por defecto es una interfaz de líneas de comando.

2. EXPLICACION DEL MANEJO PARA EJECUTAR XV6

Una vez que hayamos instalado QUEMU en Ubuntu, procederemos descargarnos el SO desde el terminal y que la máquina virtual lo ejecute. Para ellos hemos introducido este código para poder descargarlo:

git clone git://pdos.csail.mit.edu/xv6/xv6.git

Para poder ver los archivos que contiene XV6 introduciremos:

gedit Nombre_Archivo.c

Por ultimo si queremos probar XV6 en QUEMU, introduciremos:

make quemu-nox

Esto ejecutara la imagen de XV6 en QUEMU.

Video explicativo: https://www.youtube.com/watch?v=BMECOCNWc4U

3. OBJETIVO DEL PROYECTO

Una vez montado QUEMU y XV6, procederemos ver sus archivos en c y en cada uno lo clasificaremos según el componente básico de cualquier Sistema Operativo (Gestión de procesos, Administración de memoria principal, Administración de ficheros, Gestión de dispositivo de E/S)

3.1 Clasificación de los componentes básicos del SO

En esta sección, cogeremos el código de cada archivo que está compuesto el Sistema Operativo XV6 y lo clasificaremos en cada uno de los compontes básicos del SO. Para cada archivo, incluye una breve descripción sobre su funcionamiento y una parte (o completo) del código que está compuesto.

3.1.1 Clasificación sobre el Gestión de procesos.

bootmain.c:

- **Descripción:**

  Es el arranque del sistema de XV6

- **Código**

```
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum;
  for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
  }
```

init.c:

- **Descripción:**

  Es el primer programa que arrancamos a nivel usuario

- **Codigo:**

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"
char *argv[] = { "sh", 0 };
Int
main(void)
{
  int pid, wpid;
  if(open("console", O_RDWR) < 0){
    mknod("console", 1, 1);
    open("console", O_RDWR);
  }
  dup(0);  // stdout
  dup(0);  // stderr
  for(;;){
```

```
    printf(1, "init: starting sh\n");
    pid = fork();
    if(pid < 0){
      printf(1, "init: fork failed\n");
      exit();
    }
    if(pid == 0){
      exec("sh", argv);
      printf(1, "init: exec sh failed\n");
      exit();
    }
    while((wpid=wait()) >= 0 && wpid != pid)
      printf(1, "zombie!\n");
  }
}
```

ioapic.c:

- **Descripción:**

Maneja las interrupciones por Hardware con la ayuda del archivo pircirq.c.

- **Código:**

```
// The I/O APIC manages hardware interrupts for an SMP system.
// http://www.intel.com/design/chipsets/datashts/29056601.pdf
// See also picirq.c.
#include "types.h"
#include "defs.h"
#include "traps.h"
#define IOAPIC  0xFEC00000   // Default physical address of IO APIC
#define REG_ID     0x00  // Register index: ID
#define REG_VER    0x01  // Register index: version
#define REG_TABLE  0x10  // Redirection table base
// The redirection table starts at REG_TABLE and uses
// two registers to configure each interrupt.
// The first (low) register in a pair contains configuration bits.
// The second (high) register contains a bitmask telling which
// CPUs can serve that interrupt.
#define INT_DISABLED   0x00010000  // Interrupt disabled
#define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
#define INT_ACTIVELOW  0x00002000  // Active low (vs high)
#define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC ID)
volatile struct ioapic *ioapic;
// IO APIC MMIO structure: write reg, then read or write data.
struct ioapic {
  uint reg;
  uint pad[3];
  uint data;
};
static uint
```

```
ioapicread(int reg)
{
  ioapic->reg = reg;
  return ioapic->data;
}
static void
ioapicwrite(int reg, uint data)
{
  ioapic->reg = reg;
  ioapic->data = data;
}
Void
ioapicinit(void)
{
  int i, id, maxintr;
  if(!ismp)
    return;
  ioapic = (volatile struct ioapic*)IOAPIC;
  maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
  id = ioapicread(REG_ID) >> 24;
  if(id != ioapicid)
    cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
  // Mark all interrupts edge-triggered, active high, disabled,
  // and not routed to any CPUs.
  for(i = 0; i <= maxintr; i++){
    ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
    ioapicwrite(REG_TABLE+2*i+1, 0);
  }
}
Void
ioapicenable(int irq, int cpunum)
{
  if(!ismp)
    return;
  // Mark interrupt edge-triggered, active high,
  // enabled, and routed to the given cpunum,
  // which happens to be that cpu's APIC ID.
  ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
  ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
}
```

lapic.c:

- **Descripción:**

Maneja Interrupciones internas

- **Código:**

```
// The local APIC manages internal (non-I/O) interrupts.
#include "types.h"
```

```
#include "defs.h"
#include "memlayout.h"
#include "traps.h"
#include "mmu.h"
#include "x86.h"
// Local APIC registers, divided by 4 for use as uint[] indices.
#define ID      (0x0020/4)   // ID
#define VER     (0x0030/4)   // Version
#define TPR     (0x0080/4)   // Task Priority
#define EOI     (0x00B0/4)   // EOI
#define SVR     (0x00F0/4)   // Spurious Interrupt Vector
  #define ENABLE     0x00000100   // Unit Enable
#define ESR     (0x0280/4)   // Error Status
#define ICRLO   (0x0300/4)   // Interrupt Command
  #define INIT       0x00000500   // INIT/RESET
  #define STARTUP    0x00000600   // Startup IPI
  #define DELIVS     0x00001000   // Delivery status
  #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
  #define DEASSERT   0x00000000
  #define LEVEL      0x00008000   // Level triggered
  #define BCAST      0x00080000   // Send to all APICs, including self.
  #define BUSY       0x00001000
  #define FIXED      0x00000000
#define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
#define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
  #define X1         0x0000000B   // divide counts by 1
  #define PERIODIC   0x00020000   // Periodic
#define PCINT   (0x0340/4)   // Performance Counter LVT
#define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
#define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
#define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
  #define MASKED     0x00010000   // Interrupt masked
#define TICR    (0x0380/4)   // Timer Initial Count
#define TCCR    (0x0390/4)   // Timer Current Count
#define TDCR    (0x03E0/4)   // Timer Divide Configuration
volatile uint *lapic;  // Initialized in mp.c
static void
lapicw(int index, int value)
{
  lapic[index] = value;
  lapic[ID];  // wait for write to finish, by reading
}
//PAGEBREAK!
Void
lapicinit(int c)
{
  if(!lapic)
    return;
  // Enable local APIC; set spurious interrupt vector.
  lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
  // The timer repeatedly counts down at bus frequency
```

```
  // from lapic[TICR] and then issues an interrupt.
  // If xv6 cared more about precise timekeeping,
  // TICR would be calibrated using an external time source.
  lapicw(TDCR, X1);
  lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
  lapicw(TICR, 10000000);
  // Disable logical interrupt lines.
  lapicw(LINT0, MASKED);
  lapicw(LINT1, MASKED);
  // Disable performance counter overflow interrupts
  // on machines that provide that interrupt entry.
  if(((lapic[VER]>>16) & 0xFF) >= 4)
    lapicw(PCINT, MASKED);
  // Map error interrupt to IRQ_ERROR.
  lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
  // Clear error status register (requires back-to-back writes).
  lapicw(ESR, 0);
  lapicw(ESR, 0);
  // Ack any outstanding interrupts.
  lapicw(EOI, 0);
  // Send an Init Level De-Assert to synchronise arbitration ID's.
  lapicw(ICRHI, 0);
  lapicw(ICRLO, BCAST | INIT | LEVEL);
  while(lapic[ICRLO] & DELIVS)
    ;
  // Enable interrupts on the APIC (but not on the processor).
  lapicw(TPR, 0);
}
Int
cpunum(void)
{
  // Cannot call cpu when interrupts are enabled:
  // result not guaranteed to last long enough to be used!
  // Would prefer to panic but even printing is chancy here:
  // almost everything, including cprintf and panic, calls cpu,
  // often indirectly through acquire and release.
  if(readeflags()&FL_IF){
    static int n;
    if(n++ == 0)
      cprintf("cpu called from %x with interrupts enabled\n",
        __builtin_return_address(0));
  }
  if(lapic)
    return lapic[ID]>>24;
  return 0;
}
// Acknowledge interrupt.
Void
lapiceoi(void)
{
  if(lapic)
```

```
    lapicw(EOI, 0);
}
// Spin for a given number of microseconds.
// On real hardware would want to tune this dynamically.
Void
microdelay(int us)
{
}
#define IO_RTC  0x70
// Start additional processor running entry code at addr.
// See Appendix B of MultiProcessor Specification.
Void
lapicstartap(uchar apicid, uint addr)
{
  int i;
  ushort *wrv;

  // "The BSP must initialize CMOS shutdown code to 0AH
  // and the warm reset vector (DWORD based at 40:67) to point at
  // the AP startup code prior to the [universal startup algorithm]."
  outb(IO_RTC, 0xF);  // offset 0xF is shutdown code
  outb(IO_RTC+1, 0x0A);
  wrv = (ushort*)P2V((0x40<<4 | 0x67));  // Warm reset vector
  wrv[0] = 0;
  wrv[1] = addr >> 4;
  // "Universal startup algorithm."
  // Send INIT (level-triggered) interrupt to reset other CPU.
  lapicw(ICRHI, apicid<<24);
  lapicw(ICRLO, INIT | LEVEL | ASSERT);
  microdelay(200);
  lapicw(ICRLO, INIT | LEVEL);
  microdelay(100);    // should be 10ms, but too slow in Bochs!

  // Send startup IPI (twice!) to enter code.
  // Regular hardware is supposed to only accept a STARTUP
  // when it is in the halted state due to an INIT.  So the second
  // should be ignored, but it is part of the official Intel algorithm.
  // Bochs complains about the second one.  Too bad for Bochs.
  for(i = 0; i < 2; i++){
    lapicw(ICRHI, apicid<<24);
    lapicw(ICRLO, STARTUP | (addr>>12));
    microdelay(200);

  }
}
```

mkdir.c:

- **Descripción:**

Manejo de los procesadores que tenga el dispositivo.

- **Código:**

```
// Multiprocessor support
// Search memory for MP description structures.
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mp.h"
#include "x86.h"
#include "mmu.h"
#include "proc.h"
struct cpu cpus[NCPU];
static struct cpu *bcpu;
int ismp;
int ncpu;
uchar ioapicid;
Int
mpbcpu(void)
{
  return bcpu-cpus;
}
static uchar
sum(uchar *addr, int len)
{
  int i, sum;

  sum = 0;
  for(i=0; i<len; i++)
    sum += addr[i];
  return sum;
}
// Look for an MP structure in the len bytes at addr.
static struct mp*
mpsearch1(uint a, int len)
{
  uchar *e, *p, *addr;
  addr = p2v(a);
  e = addr+len;
  for(p = addr; p < e; p += sizeof(struct mp))
    if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
      return (struct mp*)p;
  return 0;
}
// Search for the MP Floating Pointer Structure, which according to the
// spec is in one of the following three locations:
```

```
// 1) in the first KB of the EBDA;
// 2) in the last KB of system base memory;
// 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
static struct mp*
mpsearch(void)
{
  uchar *bda;
  uint p;
  struct mp *mp;
  bda = (uchar *) P2V(0x400);
  if((p = ((bda[0x0F]<<8)| bda[0x0E]) << 4)){
    if((mp = mpsearch1(p, 1024)))
      return mp;
  } else {
    p = ((bda[0x14]<<8)|bda[0x13])*1024;
    if((mp = mpsearch1(p-1024, 1024)))
      return mp;
  }
  return mpsearch1(0xF0000, 0x10000);
}
// Search for an MP configuration table.  For now,
// don't accept the default configurations (physaddr == 0).
// Check for correct signature, calculate the checksum and,
// if correct, check the version.
// To do: check extended table checksum.
static struct mpconf*
mpconfig(struct mp **pmp)
{
  struct mpconf *conf;
  struct mp *mp;
  if((mp = mpsearch()) == 0 || mp->physaddr == 0)
    return 0;
  conf = (struct mpconf*) p2v((uint) mp->physaddr);
  if(memcmp(conf, "PCMP", 4) != 0)
    return 0;
  if(conf->version != 1 && conf->version != 4)
    return 0;
  if(sum((uchar*)conf, conf->length) != 0)
    return 0;
  *pmp = mp;
  return conf;
}
Void
mpinit(void)
{
  uchar *p, *e;
  struct mp *mp;
  struct mpconf *conf;
  struct mpproc *proc;
  struct mpioapic *ioapic;
  bcpu = &cpus[0];
```

```
  if((conf = mpconfig(&mp)) == 0)
    return;
  ismp = 1;
  lapic = (uint*)conf->lapicaddr;
  for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
    switch(*p){
    case MPPROC:
      proc = (struct mpproc*)p;
      if(ncpu != proc->apicid){
        cprintf("mpinit: ncpu=%d apicid=%d\n", ncpu, proc->apicid);
        ismp = 0;
      }
      if(proc->flags & MPBOOT)
        bcpu = &cpus[ncpu];
      cpus[ncpu].id = ncpu;
      ncpu++;
      p += sizeof(struct mpproc);
      continue;
    case MPIOAPIC:
      ioapic = (struct mpioapic*)p;
      ioapicid = ioapic->apicno;
      p += sizeof(struct mpioapic);
      continue;
    case MPBUS:
    case MPIOINTR:
    case MPLINTR:
      p += 8;
      continue;
    default:
      cprintf("mpinit: unknown config type %x\n", *p);
      ismp = 0;
    }
  }
  if(!ismp){
    // Didn't like what we found; fall back to no MP.
    ncpu = 1;
    lapic = 0;
    ioapicid = 0;
    return;
  }
  if(mp->imcrp){
    // Bochs doesn't support IMCR, so this doesn't run on Bochs.
    // But it would on real hardware.
    outb(0x22, 0x70);   // Select IMCR
    outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
  }
}
```

proc.c:

- **Descripción:**

Se trata del planificador del SO.

- **Código:**

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "x86.h"
#include "proc.h"
#include "spinlock.h"
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
static struct proc *initproc;
int nextpid = 1;
extern void forkret(void);
extern void trapret(void);
static void wakeup1(void *chan);
Void
pinit(void)
{
  initlock(&ptable.lock, "ptable");
}
//PAGEBREAK: 32
// Look in the process table for an UNUSED proc.
// If found, change state to EMBRYO and initialize
// state required to run in the kernel.
// Otherwise return 0.
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;
  release(&ptable.lock);
  return 0;
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  release(&ptable.lock);
  // Allocate kernel stack.
  if((p->kstack = kalloc()) == 0){
```

```
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;
  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
  return p;
}
//PAGEBREAK: 32
// Set up first user process.
Void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();
  initproc = p;
  if((p->pgdir = setupkvm(kalloc)) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S
  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");
  p->state = RUNNABLE;
}
// Grow current process's memory by n bytes.
// Return 0 on success, -1 on failure.
Int
growproc(int n)
{
  uint sz;
```

```
  sz = proc->sz;
  if(n > 0){
    if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
      return -1;
  } else if(n < 0){
    if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
      return -1;
  }
  proc->sz = sz;
  switchuvm(proc);
  return 0;
}
// Create a new process copying p as the parent.
// Sets up stack to return as if from system call.
// Caller must set state of returned proc to RUNNABLE.
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(proc->pgdir, proc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = proc->sz;
  np->parent = proc;
  *np->tf = *proc->tf;
  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;
  for(i = 0; i < NOFILE; i++)
    if(proc->ofile[i])
      np->ofile[i] = filedup(proc->ofile[i]);
  np->cwd = idup(proc->cwd);

  pid = np->pid;
  np->state = RUNNABLE;
  safestrcpy(np->name, proc->name, sizeof(proc->name));
  return pid;
}
// Exit the current process.  Does not return.
// An exited process remains in the zombie state
// until its parent calls wait() to find out it exited.
Void
exit(void)
```

```
{
  struct proc *p;
  int fd;
  if(proc == initproc)
    panic("init exiting");
  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(proc->ofile[fd]){
      fileclose(proc->ofile[fd]);
      proc->ofile[fd] = 0;
    }
  }
  iput(proc->cwd);
  proc->cwd = 0;
  acquire(&ptable.lock);
  // Parent might be sleeping in wait().
  wakeup1(proc->parent);
  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == proc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }
  // Jump into the scheduler, never to return.
  proc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
// Wait for a child process to exit and return its pid.
// Return -1 if this process has no children.
Int
wait(void)
{
  struct proc *p;
  int havekids, pid;
  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for zombie children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != proc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
```

```
        p->state = UNUSED;
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        release(&ptable.lock);
        return pid;
      }
    }
    // No point waiting if we don't have any children.
    if(!havekids || proc->killed){
      release(&ptable.lock);
      return -1;
    }
    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(proc, &ptable.lock);  //DOC: wait-sleep
  }
}
//PAGEBREAK: 42
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run
//  - swtch to start running that process
//  - eventually that process transfers control
//      via swtch back to the scheduler.
Void
scheduler(void)
{
  struct proc *p;
  for(;;){
    // Enable interrupts on this processor.
    sti();
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();
      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
    }
    release(&ptable.lock);
```

```
  }
}
// Enter scheduler.  Must hold only ptable.lock
// and have changed proc->state.
Void
sched(void)
{
  int intena;
  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(cpu->ncli != 1)
    panic("sched locks");
  if(proc->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = cpu->intena;
  swtch(&proc->context, cpu->scheduler);
  cpu->intena = intena;
}
// Give up the CPU for one scheduling round.
Void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  proc->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
// A fork child's very first scheduling by scheduler()
// will swtch here.  "Return" to user space.
Void
forkret(void)
{
  static int first = 1;
  // Still holding ptable.lock from scheduler.
  release(&ptable.lock);
  if (first) {
    // Some initialization functions must be run in the context
    // of a regular process (e.g., they call sleep), and thus cannot
    // be run from main().
    first = 0;
    initlog();
  }

  // Return to "caller", actually trapret (see allocproc).
}
// Atomically release lock and sleep on chan.
// Reacquires lock when awakened.
Void
sleep(void *chan, struct spinlock *lk)
```

```
{
  if(proc == 0)
    panic("sleep");
  if(lk == 0)
    panic("sleep without lk");
  // Must acquire ptable.lock in order to
  // change p->state and then call sched.
  // Once we hold ptable.lock, we can be
  // guaranteed that we won't miss any wakeup
  // (wakeup runs with ptable.lock locked),
  // so it's okay to release lk.
  if(lk != &ptable.lock){  //DOC: sleeplock0
    acquire(&ptable.lock);  //DOC: sleeplock1
    release(lk);
  }
  // Go to sleep.
  proc->chan = chan;
  proc->state = SLEEPING;
  sched();
  // Tidy up.
  proc->chan = 0;
  // Reacquire original lock.
  if(lk != &ptable.lock){  //DOC: sleeplock2
    release(&ptable.lock);
    acquire(lk);
  }
}
//PAGEBREAK!
// Wake up all processes sleeping on chan.
// The ptable lock must be held.
static void
wakeup1(void *chan)
{
  struct proc *p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}
// Wake up all processes sleeping on chan.
Void
wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}
// Kill the process with the given pid.
// Process won't exit until it returns
// to user space (see trap in trap.c).
Int
kill(int pid)
```

```
{
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->killed = 1;
      // Wake process from sleep if necessary.
      if(p->state == SLEEPING)
        p->state = RUNNABLE;
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
//PAGEBREAK: 36
// Print a process listing to console.  For debugging.
// Runs when user types ^P on console.
// No lock to avoid wedging a stuck machine further.
Void
procdump(void)
{
  static char *states[] = {
  [UNUSED]    "unused",
  [EMBRYO]    "embryo",
  [SLEEPING]  "sleep ",
  [RUNNABLE]  "runble",
  [RUNNING]   "run   ",
  [ZOMBIE]    "zombie"
  };
  int i;
  struct proc *p;
  char *state;
  uint pc[10];

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == UNUSED)
      continue;
    if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
      state = states[p->state];
    Else
      state = "???";
    cprintf("%d %s %s", p->pid, state, p->name);
    if(p->state == SLEEPING){
      getcallerpcs((uint*)p->context->ebp+2, pc);
      for(i=0; i<10 && pc[i] != 0; i++)
        cprintf(" %p", pc[i]);
    }
    cprintf("\n");
  }
```

```
}
```

spinlock.c:

- **Descripcion:**

Es el uso del manejo de cerradura visto en la asignatura para garantizar la exclusión mutua.

- **Codigo:**

```
// Mutual exclusion spin locks.
#include "types.h"
#include "defs.h"
#include "param.h"
#include "x86.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "spinlock.h"
Void
initlock(struct spinlock *lk, char *name)
{
  lk->name = name;
  lk->locked = 0;
  lk->cpu = 0;
}
// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
Void
acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");
  // The xchg is atomic.
  // It also serializes, so that reads after acquire are not
  // reordered before it.
  while(xchg(&lk->locked, 1) != 0)
    ;
  // Record info about lock acquisition for debugging.
  lk->cpu = cpu;
  getcallerpcs(&lk, lk->pcs);
}
// Release the lock.
Void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");
```

```
  lk->pcs[0] = 0;
  lk->cpu = 0;
  // The xchg serializes, so that reads before release are
  // not reordered after it.  The 1996 PentiumPro manual (Volume 3,
  // 7.2) says reads can be carried out speculatively and in
  // any order, which implies we need to serialize here.
  // But the 2007 Intel 64 Architecture Memory Ordering White
  // Paper says that Intel 64 and IA-32 will not move a load
  // after a store. So lock->locked = 0 would work here.
  // The xchg being asm volatile ensures gcc emits it after
  // the above assignments (and after the critical section).
  xchg(&lk->locked, 0);
  popcli();
}
// Record the current call stack in pcs[] by following the %ebp chain.
Void
getcallerpcs(void *v, uint pcs[])
{
  uint *ebp;
  int i;

  ebp = (uint*)v - 2;
  for(i = 0; i < 10; i++){
    if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff)
      break;
    pcs[i] = ebp[1];     // saved %eip
    ebp = (uint*)ebp[0]; // saved %ebp
  }
  for(; i < 10; i++)
    pcs[i] = 0;
}
// Check whether this cpu is holding the lock.
Int
holding(struct spinlock *lock)
{
  return lock->locked && lock->cpu == cpu;
}
// Pushcli/popcli are like cli/sti except that they are matched:
// it takes two popcli to undo two pushcli.  Also, if interrupts
// are off, then pushcli, popcli leaves them off.
Void
pushcli(void)
{
  int eflags;

  eflags = readeflags();
  cli();
  if(cpu->ncli++ == 0)
    cpu->intena = eflags & FL_IF;
}
Void
```

```
popcli(void)
{
  if(readeflags()&FL_IF)
    panic("popcli - interruptible");
  if(--cpu->ncli < 0)
    panic("popcli");
  if(cpu->ncli == 0 && cpu->intena)
    sti();
}
```

syscall.c:

- **Descripción:**

Maneja las funciones de llamada al sistema buscando el manejo en las bibliotecas del SO

- **Código:**

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
#include "syscall.h"
// User code makes a system call with INT T_SYSCALL.
// System call number in %eax.
// Arguments on the stack, from the user call to the C
// library system call function. The saved user %esp points
// to a saved program counter, and then the first argument.
// Fetch the int at addr from the current process.
Int
fetchint(uint addr, int *ip)
{
  if(addr >= proc->sz || addr+4 > proc->sz)
    return -1;
  *ip = *(int*)(addr);
  return 0;
}
// Fetch the nul-terminated string at addr from the current process.
// Doesn't actually copy the string - just sets *pp to point at it.
// Returns length of string, not including nul.
Int
fetchstr(uint addr, char **pp)
{
  char *s, *ep;
  if(addr >= proc->sz)
    return -1;
```

```
  *pp = (char*)addr;
  ep = (char*)proc->sz;
  for(s = *pp; s < ep; s++)
    if(*s == 0)
      return s - *pp;
  return -1;
}
// Fetch the nth 32-bit system call argument.
Int
argint(int n, int *ip)
{
  return fetchint(proc->tf->esp + 4 + 4*n, ip);
}
// Fetch the nth word-sized system call argument as a pointer
// to a block of memory of size n bytes.  Check that the pointer
// lies within the process address space.
Int
argptr(int n, char **pp, int size)
{
  int i;

  if(argint(n, &i) < 0)
    return -1;
  if((uint)i >= proc->sz || (uint)i+size > proc->sz)
    return -1;
  *pp = (char*)i;
  return 0;
}
// Fetch the nth word-sized system call argument as a string pointer.
// Check that the pointer is valid and the string is nul-terminated.
// (There is no shared writable memory, so the string can't change
// between this check and being used by the kernel.)
Int
argstr(int n, char **pp)
{
  int addr;
  if(argint(n, &addr) < 0)
    return -1;
  return fetchstr(addr, pp);
}
extern int sys_chdir(void);
extern int sys_close(void);
extern int sys_dup(void);
extern int sys_exec(void);
extern int sys_exit(void);
extern int sys_fork(void);
extern int sys_fstat(void);
extern int sys_getpid(void);
extern int sys_kill(void);
extern int sys_link(void);
extern int sys_mkdir(void);
```

```
extern int sys_mknod(void);
extern int sys_open(void);
extern int sys_pipe(void);
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
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,
};
Void
syscall(void)
{
  int num;
  num = proc->tf->eax;
  if(num >= 0 && num < SYS_open && syscalls[num]) {
    proc->tf->eax = syscalls[num]();
  } else if (num >= SYS_open && num < NELEM(syscalls) && syscalls[num]) {
    proc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            proc->pid, proc->name, num);
    proc->tf->eax = -1;
  }
}
```

Zombie.c:

- **Descripcion:**

Es la ejecución de un proceso hijo que no es recogido por el proceso padre.

- **Codigo:**

```
// Create a zombie process that
// must be reparented at exit.
#include "types.h"
#include "stat.h"
#include "user.h"
Int
main(void)
{
  if(fork() > 0)
    sleep(5);  // Let child exit before parent.
  exit();
}
```

trap.c:

- **Descripcion:**
  Contiene el contenido de todas las interrupciones del software y cuando se debe usar.

- **Código:**

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
#include "traps.h"
#include "spinlock.h"
// Interrupt descriptor table (shared by all CPUs).
struct gatedesc idt[256];
extern uint vectors[];  // in vectors.S: array of 256 entry pointers
struct spinlock tickslock;
uint ticks;
void
tvinit(void)
{
  int i;
  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
```

```
}
void
idtinit(void)
{
  lidt(idt, sizeof(idt));
}
//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(proc->killed)
      exit();
    proc->tf = tf;
    syscall();
    if(proc->killed)
      exit();
    return;
  }
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpu->id == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE:
    ideintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE+1:
    // Bochs generates spurious IDE1 interrupts.
    break;
  case T_IRQ0 + IRQ_KBD:
    kbdintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_COM1:
    uartintr();
    lapiceoi();
    break;
  case T_IRQ0 + 7:
  case T_IRQ0 + IRQ_SPURIOUS:
    cprintf("cpu%d: spurious interrupt at %x:%x\n",
            cpu->id, tf->cs, tf->eip);
    lapiceoi();
    break;
```

```
  //PAGEBREAK: 13
  default:
    if(proc == 0 || (tf->cs&3) == 0){
      // In kernel, it must be our mistake.
      cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
              tf->trapno, cpu->id, tf->eip, rcr2());
      panic("trap");
    }
    // In user space, assume process misbehaved.
    cprintf("pid %d %s: trap %d err %d on cpu %d "
            "eip 0x%x addr 0x%x--kill proc\n",
            proc->pid, proc->name, tf->trapno, tf->err, cpu->id, tf->eip,
            rcr2());
    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(proc && proc->killed && (tf->cs&3) == DPL_USER)
    exit();
  // Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to check nlock.
  if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();
  // Check if the process has been killed since we yielded
  if(proc && proc->killed && (tf->cs&3) == DPL_USER)
    exit();
}
```

uart.c:

- **Descripción:**

  Manejo de los procesos por el método FIFO

- **Código:**

```
// Intel 8250 serial port (UART).
#include "types.h"
#include "defs.h"
#include "param.h"
#include "traps.h"
#include "spinlock.h"
#include "fs.h"
#include "file.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
#define COM1    0x3f8
static int uart;    // is there a uart?
Void
uartinit(void)
{
  char *p;
```

```
  // Turn off the FIFO
  outb(COM1+2, 0);

  // 9600 baud, 8 data bits, 1 stop bit, parity off.
  outb(COM1+3, 0x80);    // Unlock divisor
  outb(COM1+0, 115200/9600);
  outb(COM1+1, 0);
  outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
  outb(COM1+4, 0);
  outb(COM1+1, 0x01);    // Enable receive interrupts.
  // If status is 0xFF, no serial port.
  if(inb(COM1+5) == 0xFF)
    return;
  uart = 1;
  // Acknowledge pre-existing interrupt conditions;
  // enable interrupts.
  inb(COM1+2);
  inb(COM1+0);
  picenable(IRQ_COM1);
  ioapicenable(IRQ_COM1, 0);

  // Announce that we're here.
  for(p="xv6...\n"; *p; p++)
    uartputc(*p);
}
void
uartputc(int c)
{
  int i;
  if(!uart)
    return;
  for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
    microdelay(10);
  outb(COM1+0, c);
}
static int
uartgetc(void)
{
  if(!uart)
    return -1;
  if(!(inb(COM1+5) & 0x01))
    return -1;
  return inb(COM1+0);
}
void
uartintr(void)
{
  consoleintr(uartgetc);
}
```

3.1.2. Clasificación sobre el Administración de memoria principal

bio.c:

- **Descripción:**

Maneja la cache de memoria las copias de contenido del bloque de disco.

- **Código:**

```
Void
binit(void)
{
  struct buf *b;
  initlock(&bcache.lock, "bcache");
//PAGEBREAK!
  // Create linked list of buffers
  bcache.head.prev = &bcache.head;
  bcache.head.next = &bcache.head;
  for(b = bcache.buf; b < bcache.buf+NBUF; b++){
    b->next = bcache.head.next;
    b->prev = &bcache.head;
    b->dev = -1;
    bcache.head.next->prev = b;
    bcache.head.next = b;
  }
}
```

Exec..c

- **Funcionamiento:**

Asigna dos paginas para cargar el programa.Una de ellas se usara de pila

- **Codigo:**

```
/ 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;
  }
  iunlockput(ip);
```

```
  ip = 0;
  // Allocate two pages at the next page boundary.
  // Make the first inaccessible.  Use the second as the user stack.
  sz = PGROUNDUP(sz);
  if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
  clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
  sp = sz;
  // Push argument strings, prepare rest of stack in ustack.
  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;
  }
```

kalloc.c:

- **Descripción:**

Asigna a memoria el proceso que se vaya a ejecutar

- **Código:**

```
// Physical memory allocator, intended to allocate
```

```
// memory for user processes, kernel stacks, page table pages,
// and pipe buffers. Allocates 4096-byte pages.
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "spinlock.h"
void freerange(void *vstart, void *vend);
extern char end[]; // first address after kernel loaded from ELF file
struct run {
  struct run *next;
};
struct {
```

```
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
} kmem;
// Initialization happens in two phases.
// 1. main() calls kinit1() while still using entrypgdir to place just
// the pages mapped by entrypgdir on free list.
// 2. main() calls kinit2() with the rest of the physical pages
// after installing a full page table that maps them on all cores.
void
kinit1(void *vstart, void *vend)
{
  initlock(&kmem.lock, "kmem");
  kmem.use_lock = 0;
  freerange(vstart, vend);
}
void
kinit2(void *vstart, void *vend)
{
  freerange(vstart, vend);
  kmem.use_lock = 1;
}
void
freerange(void *vstart, void *vend)
                                                                    {
  char *p;
  p = (char*)PGROUNDUP((uint)vstart);
  for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
    kfree(p);
}
//PAGEBREAK: 21
// Free the page of physical memory pointed at by v,
// which normally should have been returned by a
// call to kalloc().  (The exception is when
// initializing the allocator; see kinit above.)
Void
kfree(char *v)
{
  struct run *r;
  if((uint)v % PGSIZE || v < end || v2p(v) >= PHYSTOP)
    panic("kfree");
  // Fill with junk to catch dangling refs.
  memset(v, 1, PGSIZE);
  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
  if(kmem.use_lock)
    release(&kmem.lock);
}
```

```
  // Allocate one 4096-byte page of physical memory.
  // Returns a pointer that the kernel can use.
  // Returns 0 if the memory cannot be allocated.
  char*
  kalloc(void)
  {
    struct run *r;
    if(kmem.use_lock)
      acquire(&kmem.lock);
    r = kmem.freelist;
    if(r)
      kmem.freelist = r->next;
    if(kmem.use_lock)
      release(&kmem.lock);

    return (char*)r;
                                                                    }
```

log.c:

- **Descripción:**

Es el manejo que usa para el reg/memoria de los procesos.

- **Código:**

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "spinlock.h"
#include "fs.h"
#include "buf.h"
static void
write_head(void)
{
  struct buf *buf = bread(log.dev, log.start);
  struct logheader *hb = (struct logheader *) (buf->data);
  int i;
  hb->n = log.lh.n;
  for (i = 0; i < log.lh.n; i++) {
    hb->sector[i] = log.lh.sector[i];
  }
  bwrite(buf);
```

```
  brelse(buf);
}
static void
recover_from_log(void)
{
  read_head();
  install_trans(); // if committed, copy from log to disk
  log.lh.n = 0;
  write_head(); // clear the log
}
Void
begin_trans(void)
{
  acquire(&log.lock);
  while (log.busy) {
    sleep(&log, &log.lock);
  }
  log.busy = 1;
  release(&log.lock);
}
Void
commit_trans(void)
{
  if (log.lh.n > 0) {
    write_head();    // Write header to disk -- the real commit
    install_trans(); // Now install writes to home locations
    log.lh.n = 0;
    write_head();    // Erase the transaction from the log
  }

  acquire(&log.lock);
  log.busy = 0;
  wakeup(&log);
  release(&log.lock);
}
// Caller has modified b->data and is done with the buffer.
// Append the block to the log and record the block number,
// but don't write the log header (which would commit the write).
// log_write() replaces bwrite(); a typical use is:
//   bp = bread(...)
//   modify bp->data[]
//   log_write(bp)
//   brelse(bp)
Void
log_write(struct buf *b)
{
  int i;
  if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
    panic("too big a transaction");
  if (!log.busy)
    panic("write outside of trans");
```

```
  for (i = 0; i < log.lh.n; i++) {
    if (log.lh.sector[i] == b->sector)   // log absorbtion?
      break;
  }
  log.lh.sector[i] = b->sector;
  struct buf *lbuf = bread(b->dev, log.start+i+1);
  memmove(lbuf->data, b->data, BSIZE);
  bwrite(lbuf);
  brelse(lbuf);
  if (i == log.lh.n)
    log.lh.n++;
  b->flags |= B_DIRTY; // XXX prevent eviction
}
//PAGEBREAK!
// Blank page.
```

vm.c:

- **Descripción:**

Manejo de la tabla de páginas.

- **Código:**

```
#include "param.h"
#include "types.h"
#include "defs.h"
#include "x86.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "elf.h"
extern char data[];  // defined by kernel.ld
pde_t *kpgdir;  // for use in scheduler()
struct segdesc gdt[NSEGS];
// Set up CPU's kernel segment descriptors.
// Run once on entry on each CPU.
Void
seginit(void)
{
  struct cpu *c;
```

```
  // Map "logical" addresses to virtual addresses using identity map.
  // Cannot share a CODE descriptor for both kernel and user
  // because it would have to have DPL_USR, but the CPU forbids
  // an interrupt from CPL=0 to DPL=3.
  c = &cpus[cpunum()];
  c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
  c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
  c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
  c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
  // Map cpu, and curproc
  c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);
  lgdt(c->gdt, sizeof(c->gdt));
  loadgs(SEG_KCPU << 3);

  // Initialize cpu-local storage.
  cpu = c;
  proc = 0;
}
// Return the address of the PTE in page table pgdir
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;
  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)p2v(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = v2p(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}
// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
```

```
    last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
    for(;;){
      if((pte = walkpgdir(pgdir, a, 1)) == 0)
        return -1;
      if(*pte & PTE_P)
        panic("remap");
      *pte = pa | perm | PTE_P;
      if(a == last)
        break;
      a += PGSIZE;
      pa += PGSIZE;
    }
    return 0;
  }
```

umalloc.c

- **Descripción:**
  Es el asignador de memoria para los procesos.
- **Código:**

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "param.h"
// Memory allocator by Kernighan and Ritchie,
// The C programming Language, 2nd ed.  Section 8.7.
typedef long Align;
union header {
  struct {
    union header *ptr;
    uint size;
  } s;
  Align x;
};
typedef union header Header;
static Header base;
static Header *freep;
Void
free(void *ap)
{
  Header *bp, *p;
  bp = (Header*)ap - 1;
  for(p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
    if(p >= p->s.ptr && (bp > p || bp < p->s.ptr))
      break;
  if(bp + bp->s.size == p->s.ptr){
    bp->s.size += p->s.ptr->s.size;
    bp->s.ptr = p->s.ptr->s.ptr;
  } else
```

```
    bp->s.ptr = p->s.ptr;
  if(p + p->s.size == bp){
    p->s.size += bp->s.size;
    p->s.ptr = bp->s.ptr;
  } else
    p->s.ptr = bp;
  freep = p;
}
static Header*
morecore(uint nu)
{
  char *p;
  Header *hp;
  if(nu < 4096)
    nu = 4096;
  p = sbrk(nu * sizeof(Header));
  if(p == (char*)-1)
    return 0;
  hp = (Header*)p;
  hp->s.size = nu;
  free((void*)(hp + 1));
  return freep;
}
void*
malloc(uint nbytes)
{
  Header *p, *prevp;
  uint nunits;
  nunits = (nbytes + sizeof(Header) - 1)/sizeof(Header) + 1;
  if((prevp = freep) == 0){
    base.s.ptr = freep = prevp = &base;
    base.s.size = 0;
  }
  for(p = prevp->s.ptr; ; prevp = p, p = p->s.ptr){
    if(p->s.size >= nunits){
      if(p->s.size == nunits)
        prevp->s.ptr = p->s.ptr;
      else {
        p->s.size -= nunits;
        p += p->s.size;
        p->s.size = nunits;
      }
      freep = prevp;
      return (void*)(p + 1);
    }
    if(p == freep)
      if((p = morecore(nunits)) == 0)
        return 0;
  }
}
```

3.1.3 Clasificación sobre el Gestión de E/S

Console.c:

- **Descripción:**

Se trata de la Entrada y la Salida de la propia consola

- **Código:**

```
// Print to the console. only understands%d, %x, %p,%s.


Void
cprintf(char *fmt, ...)
{
  int i, c, locking;
  uint *argp;
  char *s;
  locking = cons.locking;
  if(locking)
    acquire(&cons.lock);
  if (fmt == 0)
    panic("null fmt");
  argp = (uint*)(void*)(&fmt + 1);
  for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
    if(c != '%'){
      consputc(c);
      continue;
    }
    c = fmt[++i] & 0xff;
    if(c == 0)
      break;
    switch(c){
    case 'd':
      printint(*argp++, 10, 1);
      break;
    case 'x':
    case 'p':
      printint(*argp++, 16, 0);
      break;
    case 's':
```

```
      if((s = (char*)*argp++) == 0)
        s = "(null)";
      for(; *s; s++)
        consputc(*s);
      break;
    case '%':
      consputc('%');
      break;
    default:
      // Print unknown % sequence to draw attention.
      consputc('%');
      consputc(c);
      break;
    }
  }
```

sh.c:

- **Descripción:**

Manejo de comandos de la consola

- **Código:**

```
// Shell.
#include "types.h"
#include "user.h"
#include "fcntl.h"
// Parsed command representation
#define EXEC  1
#define REDIR 2
#define PIPE  3
#define LIST  4
#define BACK  5
#define MAXARGS 10
struct cmd {
  int type;
};
struct execcmd {
  int type;
  char *argv[MAXARGS];
  char *eargv[MAXARGS];
};
struct redircmd {
  int type;
  struct cmd *cmd;
  char *file;
  char *efile;
  int mode;
  int fd;
```

```
};
struct pipecmd {
  int type;
  struct cmd *left;
  struct cmd *right;
};
struct listcmd {
  int type;
  struct cmd *left;
  struct cmd *right;
};
struct backcmd {
  int type;
  struct cmd *cmd;
};
int fork1(void);  // Fork but panics on failure.
void panic(char*);
struct cmd *parsecmd(char*);
// Execute cmd.  Never returns.
Void
runcmd(struct cmd *cmd)
{
  int p[2];
  struct backcmd *bcmd;
  struct execcmd *ecmd;
  struct listcmd *lcmd;
  struct pipecmd *pcmd;
  struct redircmd *rcmd;
  if(cmd == 0)
    exit();

  switch(cmd->type){
  default:
    panic("runcmd");
  case EXEC:
    ecmd = (struct execcmd*)cmd;
    if(ecmd->argv[0] == 0)
      exit();
    exec(ecmd->argv[0], ecmd->argv);
    printf(2, "exec %s failed\n", ecmd->argv[0]);
    break;
  case REDIR:
    rcmd = (struct redircmd*)cmd;
    close(rcmd->fd);
    if(open(rcmd->file, rcmd->mode) < 0){
      printf(2, "open %s failed\n", rcmd->file);
      exit();
    }
    runcmd(rcmd->cmd);
    break;
  case LIST:
```

```
    lcmd = (struct listcmd*)cmd;
    if(fork1() == 0)
      runcmd(lcmd->left);
    wait();
    runcmd(lcmd->right);
    break;
  case PIPE:
    pcmd = (struct pipecmd*)cmd;
    if(pipe(p) < 0)
      panic("pipe");
    if(fork1() == 0){
      close(1);
      dup(p[1]);
      close(p[0]);
      close(p[1]);
      runcmd(pcmd->left);
    }
    if(fork1() == 0){
      close(0);
      dup(p[0]);
      close(p[0]);
      close(p[1]);
      runcmd(pcmd->right);
    }
    close(p[0]);
    close(p[1]);
    wait();
    wait();
    break;

  case BACK:
    bcmd = (struct backcmd*)cmd;
    if(fork1() == 0)
      runcmd(bcmd->cmd);
    break;
  }
  exit();
}
Int
getcmd(char *buf, int nbuf)
{
  printf(2, "$ ");
  memset(buf, 0, nbuf);
  gets(buf, nbuf);
  if(buf[0] == 0) // EOF
    return -1;
  return 0;
}
```

ide.c:

- **Descripción:**

Es el driver que maneja el controlador IDE

- **Codigo:**

```
// Simple PIO-based (non-DMA) IDE driver code.
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
#include "traps.h"
#include "spinlock.h"
#include "buf.h"
#define IDE_BSY       0x80
#define IDE_DRDY      0x40
#define IDE_DF        0x20
#define IDE_ERR       0x01
#define IDE_CMD_READ  0x20
#define IDE_CMD_WRITE 0x30
// idequeue points to the buf now being read/written to the disk.
// idequeue->qnext points to the next buf to be processed.
// You must hold idelock while manipulating queue.
static struct spinlock idelock;
static struct buf *idequeue;
static int havedisk1;
static void idestart(struct buf*);
// Wait for IDE disk to become ready.
static int
idewait(int checkerr)
{
  int r;
  while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
    ;
  if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
    return -1;
  return 0;
}
Void
ideinit(void)
{
  int i;
  initlock(&idelock, "ide");
  picenable(IRQ_IDE);
  ioapicenable(IRQ_IDE, ncpu - 1);
  idewait(0);
```

```
  // Check if disk 1 is present
  outb(0x1f6, 0xe0 | (1<<4));
  for(i=0; i<1000; i++){
    if(inb(0x1f7) != 0){
      havedisk1 = 1;
      break;
    }
  }

  // Switch back to disk 0.
  outb(0x1f6, 0xe0 | (0<<4));
}
// Start the request for b.  Caller must hold idelock.
static void
idestart(struct buf *b)
{
  if(b == 0)
    panic("idestart");
  idewait(0);
  outb(0x3f6, 0);  // generate interrupt
  outb(0x1f2, 1);  // number of sectors
  outb(0x1f3, b->sector & 0xff);
  outb(0x1f4, (b->sector >> 8) & 0xff);
  outb(0x1f5, (b->sector >> 16) & 0xff);
  outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
  if(b->flags & B_DIRTY){
    outb(0x1f7, IDE_CMD_WRITE);
    outsl(0x1f0, b->data, 512/4);
  } else {
    outb(0x1f7, IDE_CMD_READ);
  }
}
// Interrupt handler.
Void
ideintr(void)
{
  struct buf *b;
  // First queued buffer is the active request.
  acquire(&idelock);
  if((b = idequeue) == 0){
    release(&idelock);
    // cprintf("spurious IDE interrupt\n");
    return;
  }
  idequeue = b->qnext;
  // Read data if needed.
  if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
    insl(0x1f0, b->data, 512/4);

  // Wake process waiting for this buf.
  b->flags |= B_VALID;
```

```
  b->flags &= ~B_DIRTY;
  wakeup(b);

  // Start disk on next buf in queue.
  if(idequeue != 0)
    idestart(idequeue);
  release(&idelock);
}
//PAGEBREAK!
// Sync buf with disk.
// If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
// Else if B_VALID is not set, read buf from disk, set B_VALID.
Void
iderw(struct buf *b)
{
  struct buf **pp;
  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 != 0 && !havedisk1)
    panic("iderw: ide disk 1 not present");
  acquire(&idelock);  //DOC: acquire-lock
  // Append b to idequeue.
  b->qnext = 0;
  for(pp=&idequeue; *pp; pp=&(*pp)->qnext)  //DOC: insert-queue
    ;
  *pp = b;

  // Start disk if necessary.
  if(idequeue == b)
    idestart(b);

  // Wait for request to finish.
  while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
    sleep(b, &idelock);
  }
  release(&idelock);
}
```

3.1.4 Clasificación sobre el Administración de ficheros

file.c:

- **Descripción:**

Obtiene información del fichero que se vaya a ejecutar.

- **Código:**

```
// File descriptors
//
#include "types.h"
#include "defs.h"
#include "param.h"
#include "fs.h"
#include "file.h"
#include "spinlock.h"
struct devsw devsw[NDEV];
struct {
  struct spinlock lock;
  struct file file[NFILE];
} ftable;
Void
fileinit(void)
{
  initlock(&ftable.lock, "ftable");
}
// Allocate a file structure.
struct file*
filealloc(void)
{
  struct file *f;
  acquire(&ftable.lock);
  for(f = ftable.file; f < ftable.file + NFILE; f++){
    if(f->ref == 0){
      f->ref = 1;
      release(&ftable.lock);
      return f;
    }
  }
  release(&ftable.lock);
  return 0;
}
// Increment ref count for file f.
struct file*
filedup(struct file *f)
{
  acquire(&ftable.lock);
  if(f->ref < 1)
    panic("filedup");
  f->ref++;
  release(&ftable.lock);
  return f;
}
// Close file f.  (Decrement ref count, close when reaches 0.)
```

```
Void
fileclose(struct file *f)
{
  struct file ff;
  acquire(&ftable.lock);
  if(f->ref < 1)
    panic("fileclose");
  if(--f->ref > 0){
    release(&ftable.lock);
    return;
  }
  ff = *f;
  f->ref = 0;
  f->type = FD_NONE;
  release(&ftable.lock);

  if(ff.type == FD_PIPE)
    pipeclose(ff.pipe, ff.writable);
  else if(ff.type == FD_INODE){
    begin_trans();
    iput(ff.ip);
    commit_trans();
  }
}
// Get metadata about file f.
Int
filestat(struct file *f, struct stat *st)
{
  if(f->type == FD_INODE){
    ilock(f->ip);
    stati(f->ip, st);
    iunlock(f->ip);
    return 0;
  }
  return -1;
}
// Read from file f.
Int
fileread(struct file *f, char *addr, int n)
{
  int r;
  if(f->readable == 0)
    return -1;
  if(f->type == FD_PIPE)
    return piperead(f->pipe, addr, n);
  if(f->type == FD_INODE){
    ilock(f->ip);
    if((r = readi(f->ip, addr, f->off, n)) > 0)
      f->off += r;
    iunlock(f->ip);
    return r;
```

```
  }
  panic("fileread");
}
//PAGEBREAK!
// Write to file f.
Int
filewrite(struct file *f, char *addr, int n)
{
  int r;
  if(f->writable == 0)
    return -1;
  if(f->type == FD_PIPE)
    return pipewrite(f->pipe, addr, n);
  if(f->type == FD_INODE){
    // write a few blocks at a time to avoid exceeding
    // the maximum log transaction size, including
    // i-node, indirect block, allocation blocks,
    // and 2 blocks of slop for non-aligned writes.
    // this really belongs lower down, since writei()
    // might be writing a device like the console.
    int max = ((LOGSIZE-1-1-2) / 2) * 512;
    int i = 0;
    while(i < n){
      int n1 = n - i;
      if(n1 > max)
        n1 = max;
      begin_trans();
      ilock(f->ip);
      if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
        f->off += r;
      iunlock(f->ip);
      commit_trans();
      if(r < 0)
        break;
      if(r != n1)
        panic("short filewrite");
      i += r;
    }
    return i == n ? n : -1;
  }
  panic("filewrite");
}
```

sysfile.c

- **Descripción:**

Es una llamada al sistema para el manejo de fichero

- **Código:**

```
//
// File-system system calls.
// Mostly argument checking, since we don't trust
// user code, and calls into file.c and fs.c.
//
#include "types.h"
#include "defs.h"
#include "param.h"
#include "stat.h"
#include "mmu.h"
#include "proc.h"
#include "fs.h"
#include "file.h"
#include "fcntl.h"
// Fetch the nth word-sized system call argument as a file descriptor
// and return both the descriptor and the corresponding struct file.
static int
argfd(int n, int *pfd, struct file **pf)
{
  int fd;
  struct file *f;
  if(argint(n, &fd) < 0)
    return -1;
  if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
    return -1;
  if(pfd)
    *pfd = fd;
  if(pf)
    *pf = f;
  return 0;
}
// Allocate a file descriptor for the given file.
// Takes over file reference from caller on success.
static int
fdalloc(struct file *f)
{
  int fd;
  for(fd = 0; fd < NOFILE; fd++){
    if(proc->ofile[fd] == 0){
      proc->ofile[fd] = f;
      return fd;
    }
  }
  return -1;
}
Int
sys_dup(void)
{
  struct file *f;
```

```
  int fd;

  if(argfd(0, 0, &f) < 0)
    return -1;
  if((fd=fdalloc(f)) < 0)
    return -1;
  filedup(f);
  return fd;
}
Int
sys_read(void)
{
  struct file *f;
  int n;
  char *p;
  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return fileread(f, p, n);
}
Int
sys_write(void)
{
  struct file *f;
  int n;
  char *p;
  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return filewrite(f, p, n);
}
Int
sys_close(void)
{
  int fd;
  struct file *f;

  if(argfd(0, &fd, &f) < 0)
    return -1;
  proc->ofile[fd] = 0;
  fileclose(f);
  return 0;
}
Int
sys_fstat(void)
{
  struct file *f;
  struct stat *st;

  if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
    return -1;
  return filestat(f, st);
}
```

```
// Create the path new as a link to the same inode as old.
Int
sys_link(void)
{
  char name[DIRSIZ], *new, *old;
  struct inode *dp, *ip;
  if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
    return -1;
  if((ip = namei(old)) == 0)
    return -1;
  begin_trans();
  ilock(ip);
  if(ip->type == T_DIR){
    iunlockput(ip);
    commit_trans();
    return -1;
  }
  ip->nlink++;
  iupdate(ip);
  iunlock(ip);
  if((dp = nameiparent(new, name)) == 0)
    goto bad;
  ilock(dp);
  if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
    iunlockput(dp);
    goto bad;
  }
  iunlockput(dp);
  iput(ip);
  commit_trans();
  return 0;
bad:
  ilock(ip);
  ip->nlink--;
  iupdate(ip);
  iunlockput(ip);
  commit_trans();
  return -1;
}
// Is the directory dp empty except for "." and ".." ?
static int
isdirempty(struct inode *dp)
{
  int off;
  struct dirent de;
  for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
    if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("isdirempty: readi");
    if(de.inum != 0)
      return 0;
  }
```

```
  return 1;
}
```

usertests.c

- **Descripción:**

Hace el manejo total del administrador de archivos

- **Código:**

```
/ More file system tests
// two processes write to the same file descriptor
// is the offset shared? does inode locking work?
Void
sharedfd(void)
{
  int fd, pid, i, n, nc, np;
  char buf[10];
  printf(1, "sharedfd test\n");
  unlink("sharedfd");
  fd = open("sharedfd", O_CREATE|O_RDWR);
  if(fd < 0){
    printf(1, "fstests: cannot open sharedfd for writing");
    return;
  }
  pid = fork();
  memset(buf, pid==0?'c':'p', sizeof(buf));
  for(i = 0; i < 1000; i++){
    if(write(fd, buf, sizeof(buf)) != sizeof(buf)){
      printf(1, "fstests: write sharedfd failed\n");
      break;
    }
  }
  if(pid == 0)
    exit();
  Else
    wait();
  close(fd);
  fd = open("sharedfd", 0);
  if(fd < 0){
    printf(1, "fstests: cannot open sharedfd for reading\n");
    return;
  }
  nc = np = 0;
  while((n = read(fd, buf, sizeof(buf))) > 0){
    for(i = 0; i < sizeof(buf); i++){
      if(buf[i] == 'c')
        nc++;
      if(buf[i] == 'p')
        np++;
```

```
    }
  }
  close(fd);
  unlink("sharedfd");
  if(nc == 10000 && np == 10000){
    printf(1, "sharedfd ok\n");
  } else {
    printf(1, "sharedfd oops %d %d\n", nc, np);
    exit();
  }
}
// two processes write two different files at the same
// time, to test block allocation.
Void
twofiles(void)
{
  int fd, pid, i, j, n, total;
  char *fname;
  printf(1, "twofiles test\n");
  unlink("f1");
  unlink("f2");
  pid = fork();
  if(pid < 0){
    printf(1, "fork failed\n");
    exit();
  }
  fname = pid ? "f1" : "f2";
  fd = open(fname, O_CREATE | O_RDWR);
  if(fd < 0){
    printf(1, "create failed\n");
    exit();
  }
  memset(buf, pid?'p':'c', 512);
  for(i = 0; i < 12; i++){
    if((n = write(fd, buf, 500)) != 500){
      printf(1, "write failed %d\n", n);
      exit();
    }
  }
  close(fd);
  if(pid)
    wait();
  Else
    exit();
  for(i = 0; i < 2; i++){
    fd = open(i?"f1":"f2", 0);
    total = 0;
    while((n = read(fd, buf, sizeof(buf))) > 0){
      for(j = 0; j < n; j++){
        if(buf[j] != (i?'p':'c')){
          printf(1, "wrong char\n");
```

```
        exit();
      }
    }
    total += n;
  }
  close(fd);
  if(total != 12*500){
    printf(1, "wrong length %d\n", total);
    exit();
  }
 }
 unlink("f1");
 unlink("f2");
 printf(1, "twofiles ok\n");
}
// two processes create and delete different files in same directory
Void
createdelete(void)
{
 enum { N = 20 };
 int pid, i, fd;
 char name[32];
 printf(1, "createdelete test\n");
 pid = fork();
 if(pid < 0){
  printf(1, "fork failed\n");
  exit();
 }
 name[0] = pid ? 'p' : 'c';
 name[2] = '\0';
 for(i = 0; i < N; i++){
  name[1] = '0' + i;
  fd = open(name, O_CREATE | O_RDWR);
  if(fd < 0){
    printf(1, "create failed\n");
    exit();
  }
  close(fd);
  if(i > 0 && (i % 2 ) == 0){
    name[1] = '0' + (i / 2);
    if(unlink(name) < 0){
      printf(1, "unlink failed\n");
      exit();
    }
  }
 }
 if(pid==0)
  exit();
 Else
  wait();
 for(i = 0; i < N; i++){
```

```
    name[0] = 'p';
    name[1] = '0' + i;
    fd = open(name, 0);
    if((i == 0 || i >= N/2) && fd < 0){
      printf(1, "oops createdelete %s didn't exist\n", name);
      exit();
    } else if((i >= 1 && i < N/2) && fd >= 0){
      printf(1, "oops createdelete %s did exist\n", name);
      exit();
    }
    if(fd >= 0)
      close(fd);
    name[0] = 'c';
    name[1] = '0' + i;
    fd = open(name, 0);
    if((i == 0 || i >= N/2) && fd < 0){
      printf(1, "oops createdelete %s didn't exist\n", name);
      exit();
    } else if((i >= 1 && i < N/2) && fd >= 0){
      printf(1, "oops createdelete %s did exist\n", name);
      exit();
    }
    if(fd >= 0)
      close(fd);
  }
  for(i = 0; i < N; i++){
    name[0] = 'p';
    name[1] = '0' + i;
    unlink(name);
    name[0] = 'c';
    unlink(name);
  }
  printf(1, "createdelete ok\n");
}
```

Fs.c:

- **Descripción:**

Implementacion del sistema de archivos

- **Código:**

```
// File system
implementation.  Five
```

```
layers:
                                  //   + Blocks: allocator for raw disk blocks.
//   + Log: crash recovery for multi-step updates.
//   + Files: inode allocator, reading, writing, metadata.
//   + Directories: inode with special contents (list of other
inodes!)
//   + Names: paths like /usr/rtm/xv6/fs.c for convenient
naming.
//
// This file contains the low-level file system manipulation
// routines.  The (higher-level) system call implementations
// are in sysfile.c.
#include "types.h"
#include "defs.h"
#include "param.h"
#include "stat.h"
#include "mmu.h"
#include "proc.h"
#include "spinlock.h"
#include "buf.h"
#include "fs.h"
#include "file.h"
#define min(a, b) ((a) < (b) ? (a) : (b))
static void itrunc(struct inode*);
// Read the super block.
void
readsb(int dev, struct superblock *sb)
{
  struct buf *bp;

  bp = bread(dev, 1);
  memmove(sb, bp->data, sizeof(*sb));
  brelse(bp);
}
// Zero a block.
static void
bzero(int dev, int bno)
{
  struct buf *bp;

  bp = bread(dev, bno);
  memset(bp->data, 0, BSIZE);
  log_write(bp);
  brelse(bp);
}
```

Log.c

- **Descripción:**

Manejo de los registros para el administrador de ficheros.

- **Código:**

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "spinlock.h"
#include "fs.h"
#include "buf.h"
// Simple logging. Each system call that might write the file system
// should be surrounded with begin_trans() and commit_trans() calls.
//
// The log holds at most one transaction at a time. Commit forces
// the log (with commit record) to disk, then installs the affected
// blocks to disk, then erases the log. begin_trans() ensures that
// only one system call can be in a transaction; others must wait.
//
// Allowing only one transaction at a time means that the file
// system code doesn't have to worry about the possibility of
// one transaction reading a block that another one has modified,
// for example an i-node block.
//
// Read-only system calls don't need to use transactions, though
// this means that they may observe uncommitted data. I-node and
// buffer locks prevent read-only calls from seeing inconsistent data.
//
// The log is a physical re-do log containing disk blocks.
// The on-disk log format:
//   header block, containing sector #s for block A, B, C, ...
//   block A
//   block B
//   block C
//   ...
// Log appends are synchronous.
// Contents of the header block, used for both the on-disk header block
// and to keep track in memory of logged sector #s before commit.
struct logheader {
  int n;
  int sector[LOGSIZE];
};
struct log {
  struct spinlock lock;
  int start;
  int size;
  int busy; // a transaction is active
  int dev;
  struct logheader lh;
};
```

```
struct log log;
static void recover_from_log(void);
void
initlog(void)
{
  if (sizeof(struct logheader) >= BSIZE)
    panic("initlog: too big logheader");
  struct superblock sb;
  initlock(&log.lock, "log");
  readsb(ROOTDEV, &sb);
  log.start = sb.size - sb.nlog;
  log.size = sb.nlog;
  log.dev = ROOTDEV;
  recover_from_log();
}
// Copy committed blocks from log to their home location
static void
install_trans(void)
{
  int tail;
  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
    struct buf *dbuf = bread(log.dev, log.lh.sector[tail]); // read dst
    memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
    bwrite(dbuf);  // write dst to disk
    brelse(lbuf);
    brelse(dbuf);
  }
}
// Read the log header from disk into the in-memory log header
static void
read_head(void)
{
  struct buf *buf = bread(log.dev, log.start);
  struct logheader *lh = (struct logheader *) (buf->data);
  int i;
  log.lh.n = lh->n;
  for (i = 0; i < log.lh.n; i++) {
    log.lh.sector[i] = lh->sector[i];
  }
  brelse(buf);
}
// Write in-memory log header to disk.
// This is the true point at which the
// current transaction commits.
static void
write_head(void)
{
  struct buf *buf = bread(log.dev, log.start);
  struct logheader *hb = (struct logheader *) (buf->data);
  int i;
```

```
  hb->n = log.lh.n;
  for (i = 0; i < log.lh.n; i++) {
    hb->sector[i] = log.lh.sector[i];
  }
  bwrite(buf);
  brelse(buf);
}
static void
recover_from_log(void)
{
  read_head();
  install_trans(); // if committed, copy from log to disk
  log.lh.n = 0;
  write_head(); // clear the log
}
void
begin_trans(void)
{
  acquire(&log.lock);
  while (log.busy) {
    sleep(&log, &log.lock);
  }
  log.busy = 1;
  release(&log.lock);
}
void
commit_trans(void)
{
  if (log.lh.n > 0) {
    write_head();    // Write header to disk -- the real commit
    install_trans(); // Now install writes to home locations
    log.lh.n = 0;
    write_head();    // Erase the transaction from the log
  }

  acquire(&log.lock);
  log.busy = 0;
  wakeup(&log);
  release(&log.lock);
}
// Caller has modified b->data and is done with the buffer.
// Append the block to the log and record the block number,
// but don't write the log header (which would commit the write).
// log_write() replaces bwrite(); a typical use is:
//   bp = bread(...)
//   modify bp->data[]
//   log_write(bp)
//   brelse(bp)
void
log_write(struct buf *b)
{
```

```
  int i;
  if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
    panic("too big a transaction");
  if (!log.busy)
    panic("write outside of trans");
  for (i = 0; i < log.lh.n; i++) {
    if (log.lh.sector[i] == b->sector)   // log absorbtion?
      break;
  }
  log.lh.sector[i] = b->sector;
  struct buf *lbuf = bread(b->dev, log.start+i+1);
  memmove(lbuf->data, b->data, BSIZE);
  bwrite(lbuf);
  brelse(lbuf);
  if (i == log.lh.n)
    log.lh.n++;
  b->flags |= B_DIRTY; // XXX prevent eviction
}
//PAGEBREAK!
// Blank page.
```

4. CONCLUSIÓN

El trabajo realizado en este proyecto nos demuestra como funciona realmente un Sistema Operativo partiendo de los componentes básicos que están compuestos a través de los archivos que esta compuesto.