

```

typedef unsigned int  uint;
typedef unsigned short ushort;
typedef unsigned char uchar;
typedef uint pde_t;

```

```

struct buf;
struct context;
struct file;
struct inode;
5 struct pipe;
struct proc;
struct spinlock;
struct stat;
struct superblock;

10 // bio.c
void      binit(void);
struct buf* bread(uint, uint);
void      brelse(struct buf*);
15 void      bwrite(struct buf*);

// console.c
void      consoleinit(void);
void      cprintf(char*, ...);
20 void      consoleintr(int (*)(void));
void      panic(char*) __attribute__((noreturn));

// exec.c
int       exec(char*, char**);

25 // file.c
struct file* filealloc(void);
void      fileclose(struct file*);
struct file* filedup(struct file*);
void      fileinit(void);
30 int      fileread(struct file*, char*, int n);
int      filestat(struct file*, struct stat*);
int      filewrite(struct file*, char*, int n);

35 // fs.c
void      readsb(int dev, struct superblock *sb);
int      dirlink(struct inode*, char*, uint);
struct inode* dirlookup(struct inode*, char*, uint*);
struct inode* ialloc(uint, short);
40 struct inode* idup(struct inode*);
void      iinit(void);
void      ilock(struct inode*);
void      iput(struct inode*);
void      iunlock(struct inode*);
void      iunlockput(struct inode*);
45 void      iupdate(struct inode*);
int      namecmp(const char*, const char*);
struct inode* namei(char*);
struct inode* nameiparent(char*, char*);
50 int      readi(struct inode*, char*, uint, uint);
void      stati(struct inode*, struct stat*);
int      writei(struct inode*, char*, uint, uint);

// ide.c
55 void      ideinit(void);
void      ideintr(void);
void      iderw(struct buf*);

// ioapic.c
60 void      ioapicenable(int irq, int cpu);
extern uchar ioapicid;
void      ioapicinit(void);

// kalloc.c

```

```

65 char*      kalloc(void);
void      kfree(char*);
void      kinit1(void*, void*);
void      kinit2(void*, void*);

70 // kbd.c
void      kbdtintr(void);

// lapic.c
int      cpunum(void);
75 extern volatile uint* lapic;
void      lapiceoi(void);
void      lapicinit(void);
void      lapicstartap(uchar, uint);
void      microdelay(int);

80 // log.c
void      initlog(void);
void      log_write(struct buf*);
void      begin_trans();
void      commit_trans();

85 // mp.c
extern int ismp;
extern int mpbcpu(void);
90 void      mpinit(void);
void      mpstartthem(void);

// picirq.c
void      picenable(int);
95 void      picinit(void);

// pipe.c
int      pipealloc(struct file**, struct file**);
void      pipeclose(struct pipe*, int);
100 int      piperead(struct pipe*, char*, int);
int      pipewrite(struct pipe*, char*, int);

//PAGEBREAK: 16
// proc.c
105 struct proc* copyproc(struct proc*);
void      exit(void);
int      fork(void);
int      growproc(int);
int      kill(int);
110 void      pinit(void);
void      procdump(void);
void      scheduler(void) __attribute__((noreturn));
void      sched(void);
void      sleep(void*, struct spinlock*);
115 void      userinit(void);
int      wait(void);
void      wakeup(void*);
void      yield(void);

120 // swtch.S
void      swtch(struct context**, struct context*);

// spinlock.c
125 void      acquire(struct spinlock*);
void      getcallerpcs(void*, uint*);
int      holding(struct spinlock*);
void      initlock(struct spinlock*, char*);
void      release(struct spinlock*);

```

```

void      pushcli(void);
130 void      popcli(void);

// string.c
int      memcmp(const void*, const void*, uint);
void*    memmove(void*, const void*, uint);
135 void*    memset(void*, int, uint);
char*    safestrcpy(char*, const char*, int);
int      strlen(const char*);
int      strncmp(const char*, const char*, uint);
140 char*    strncpy(char*, const char*, int);

// syscall.c
int      argint(int, int*);
int      argptr(int, char**, int);
int      argstr(int, char**);
145 int      fetchint(uint, int*);
int      fetchstr(uint, char**);
void      syscall(void);

// timer.c
150 void      timerinit(void);

// trap.c
void      idtinit(void);
extern uint ticks;
155 void      tvinit(void);
extern struct spinlock tickslock;

// uart.c
void      uartinit(void);
void      uartintr(void);
160 void      uartputc(int);

// vm.c
void      seginit(void);
void      kmalloc(void);
void      vmenable(void);
165 pde_t*   setupkvm(void);
char*    uvazka(pde_t*, char*);
int      allocuv(pde_t*, uint, uint);
int      deallocuv(pde_t*, uint, uint);
void      freevm(pde_t*);
void      inituvm(pde_t*, char*, uint);
170 void      loaduvm(pde_t*, char*, struct inode*, uint, uint);
pde_t*   copyuvm(pde_t*, uint);
void      switchuvm(struct proc*);
void      switchkvm(void);
int      copyout(pde_t*, uint, void*, uint);
void      clearpteu(pde_t *pgdir, char *uva);

180 // number of elements in fixed-size array
#define NELEM(x) (sizeof(x)/sizeof(x[0]))

```

```

param.h                                     Page 1/1
#define NPROC 64 // maximum number of processes
#define KSTACKSIZE 4096 // size of per-process kernel stack
#define NCPU 8 // maximum number of CPUs
#define NOFILE 16 // open files per process
5 #define NFILE 100 // open files per system
#define NBUF 10 // size of disk block cache
#define NNODE 50 // maximum number of active i-nodes
#define NDEV 10 // maximum major device number
10 #define ROOTDEV 1 // device number of file system root disk
#define MAXARG 32 // max exec arguments
#define LOGSIZE 10 // max data sectors in on-disk log

```

```

proc.h                                       Page 1/2
// Segments in proc->gdt.
#define NSEGS 7

// Per-CPU state
5 struct cpu {
    uchar id; // Local APIC ID; index into cpus[] below
    struct context *scheduler; // switch() here to enter scheduler
    struct taskstate ts; // Used by x86 to find stack for interrupt
    struct segdesc gdt[NSEGS]; // x86 global descriptor table
10 volatile uint started; // Has the CPU started?
    int ncli; // Depth of pushcli nesting.
    int intena; // Were interrupts enabled before pushcli?

    // Cpu-local storage variables; see below
15 struct cpu *cpu;
    struct proc *proc; // The currently-running process.
};

extern struct cpu cpus[NCPU];
20 extern int ncpu;

// Per-CPU variables, holding pointers to the
// current cpu and to the current process.
// The asm suffix tells gcc to use "%gs:0" to refer to cpu
// and "%gs:4" to refer to proc. seginit sets up the
25 // %gs segment register so that %gs refers to the memory
// holding those two variables in the local cpu's struct cpu.
// This is similar to how thread-local variables are implemented
// in thread libraries such as Linux pthreads.
30 extern struct cpu *cpu asm("%gs:0"); // &cpus[cpunum()]
extern struct proc *proc asm("%gs:4"); // cpus[cpunum()].proc

//PAGEBREAK: 17
// Saved registers for kernel context switches.
35 // Don't need to save all the segment registers (%cs, etc),
// because they are constant across kernel contexts.
// Don't need to save %eax, %ecx, %edx, because the
// x86 convention is that the caller has saved them.
// Contexts are stored at the bottom of the stack they
40 // describe; the stack pointer is the address of the context.
// The layout of the context matches the layout of the stack in switch.S
// at the "Switch stacks" comment. Switch doesn't save eip explicitly,
// but it is on the stack and allocproc() manipulates it.
struct context {
45     uint edi;
    uint esi;
    uint ebx;
    uint ebp;
    uint eip;
50 };

enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
55 struct proc {
    uint sz; // Size of process memory (bytes)
    pde_t *pgdir; // Page table
    char *kstack; // Bottom of kernel stack for this process
    enum procstate state; // Process state
60 volatile int pid; // Process ID
    struct proc *parent; // Parent process
    struct trapframe *tf; // Trap frame for current syscall
    struct context *context; // switch() here to run process
    void *chan; // If non-zero, sleeping on chan

```

```

proc.h                                       Page 2/2
65 int killed; // If non-zero, have been killed
struct file *ofile[NOFILE]; // Open files
struct inode *cwd; // Current directory
char name[16]; // Process name (debugging)
};
70 // Process memory is laid out contiguously, low addresses first:
// text
// original data and bss
// fixed-size stack
75 // expandable heap

```

```

memlayout.h                                 Page 1/1
// Memory layout

#define EXTMEM 0x100000 // Start of extended memory
#define PHYSTOP 0xE000000 // Top physical memory
5 #define DEVSPACE 0xF000000 // Other devices are at high addresses

// Key addresses for address space layout (see kmap in vm.c for layout)
#define KERNBASE 0x80000000 // First kernel virtual address
10 #define KERNLINK (KERNBASE+EXTMEM) // Address where kernel is linked

#ifdef __ASSEMBLER__
static inline uint v2p(void *a) { return ((uint) (a)) - KERNBASE; }
15 static inline void *p2v(uint a) { return (void *) ((a) + KERNBASE); }
#endif

#define V2P(a) (((uint) (a)) - KERNBASE)
#define P2V(a) (((void *) (a)) + KERNBASE)

20 #define V2P_WO(x) ((x) - KERNBASE) // same as V2P, but without casts
#define P2V_WO(x) ((x) + KERNBASE) // same as P2V, but without casts

```

```

// This file contains definitions for the
// x86 memory management unit (MMU).

// Eflags register
5 #define FL_CF 0x00000001 // Carry Flag
#define FL_PF 0x00000004 // Parity Flag
#define FL_AF 0x00000010 // Auxiliary carry Flag
#define FL_ZF 0x00000040 // Zero Flag
10 #define FL_SF 0x00000080 // Sign Flag
#define FL_TF 0x00000100 // Trap Flag
#define FL_IF 0x00000200 // Interrupt Enable
#define FL_DF 0x00000400 // Direction Flag
#define FL_OF 0x00000800 // Overflow Flag
15 #define FL_IOPL_MASK 0x00003000 // I/O Privilege Level bitmask
#define FL_IOPL_0 0x00000000 // IOPL == 0
#define FL_IOPL_1 0x00001000 // IOPL == 1
#define FL_IOPL_2 0x00002000 // IOPL == 2
#define FL_IOPL_3 0x00003000 // IOPL == 3
20 #define FL_NT 0x00004000 // Nested Task
#define FL_RF 0x00010000 // Resume Flag
#define FL_VM 0x00020000 // Virtual 8086 mode
#define FL_AC 0x00040000 // Alignment Check
#define FL_VIF 0x00080000 // Virtual Interrupt Flag
25 #define FL_VIP 0x00100000 // Virtual Interrupt Pending
#define FL_ID 0x00200000 // ID flag

// Control Register flags
#define CR0_PE 0x00000001 // Protection Enable
#define CR0_MP 0x00000002 // Monitor coProcessor
30 #define CR0_EM 0x00000004 // Emulation
#define CR0_TS 0x00000008 // Task Switched
#define CR0_ET 0x00000010 // Extension Type
#define CR0_NE 0x00000020 // Numeric Error
#define CR0_WP 0x00010000 // Write Protect
35 #define CR0_AM 0x00040000 // Alignment Mask
#define CR0_NW 0x00200000 // Not Writethrough
#define CR0_CD 0x00400000 // Cache Disable
#define CR0_PG 0x00800000 // Paging

40 #define CR4_PSE 0x00000010 // Page size extension

#define SEG_KCODE 1 // kernel code
#define SEG_KDATA 2 // kernel data+stack
#define SEG_KCPU 3 // kernel per-cpu data
45 #define SEG_UCODE 4 // user code
#define SEG_UDATA 5 // user data+stack
#define SEG_TSS 6 // this process's task state

//PAGEBREAK!
50 #ifndef __ASSEMBLER__
// Segment Descriptor
struct segdesc {
    uint lim_15_0 : 16; // Low bits of segment limit
    uint base_15_0 : 16; // Low bits of segment base address
55 #define base_23_16 : 8; // Middle bits of segment base address
    uint type : 4; // Segment type (see STS constants)
    uint s : 1; // 0 = system, 1 = application
    uint dpl : 2; // Descriptor Privilege Level
    uint p : 1; // Present
60 #define lim_19_16 : 4; // High bits of segment limit
    uint avl : 1; // Unused (available for software use)
    uint rsv1 : 1; // Reserved
    uint db : 1; // 0 = 16-bit segment, 1 = 32-bit segment
    uint g : 1; // Granularity: limit scaled by 4K when set
};

```

```

65 uint base_31_24 : 8; // High bits of segment base address
};

// Normal segment
70 #define SEG(type, base, lim, dpl) (struct segdesc) \
{ ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff, \
  ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1, \
  (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
#define SEG16(type, base, lim, dpl) (struct segdesc) \
{ (lim) & 0xffff, (uint)(base) & 0xffff, \
75 { (uint)(base) >> 16) & 0xff, type, 1, dpl, 1, \
  (uint)(lim) >> 16, 0, 0, 1, 0, (uint)(base) >> 24 }
#endif

#define DPL_USER 0x3 // User DPL

80 // Application segment type bits
#define STA_X 0x8 // Executable segment
#define STA_E 0x4 // Expand down (non-executable segments)
#define STA_C 0x4 // Conforming code segment (executable only)
85 #define STA_W 0x2 // Writable (non-executable segments)
#define STA_R 0x2 // Readable (executable segments)
#define STA_A 0x1 // Accessed

// System segment type bits
90 #define STS_T16A 0x1 // Available 16-bit TSS
#define STS_LDT 0x2 // Local Descriptor Table
#define STS_T16B 0x3 // Busy 16-bit TSS
#define STS_CG16 0x4 // 16-bit Call Gate
95 #define STS_TG 0x5 // Task Gate / Coum Transitions
#define STS_IG16 0x6 // 16-bit Interrupt Gate
#define STS_TG16 0x7 // 16-bit Trap Gate
#define STS_T32A 0x9 // Available 32-bit TSS
#define STS_T32B 0xB // Busy 32-bit TSS
100 #define STS_CG32 0xC // 32-bit Call Gate
#define STS_IG32 0xE // 32-bit Interrupt Gate
#define STS_TG32 0xF // 32-bit Trap Gate

// A virtual address 'la' has a three-part structure as follows:
105 // +-----10-----+-----10-----+-----12-----+
// | Page Directory | Page Table | Offset within Page |
// | Index | Index | |
// +-----PDX(va)---+---PTX(va)---+

110 // page directory index
#define PDX(va) (((uint)(va) >> PDXSHIFT) & 0x3FF)

// page table index
115 #define PTX(va) (((uint)(va) >> PTXSHIFT) & 0x3FF)

// construct virtual address from indexes and offset
#define PGADDR(d, t, o) (((uint)(d) << PDXSHIFT | (t) << PTXSHIFT | (o)))

120 // Page directory and page table constants.
#define NPENTRIES 1024 // # directory entries per page directory
#define NPTEENTRIES 1024 // # PTEs per page table
#define PGSIZE 4096 // bytes mapped by a page

125 #define PGSHIFT 12 // log2(PGSIZE)
#define PTXSHIFT 12 // offset of PTX in a linear address
#define PDXSHIFT 22 // offset of PDX in a linear address

```

```

130 #define PGROUNDUP(sz) (((sz)+PGSIZE-1) & ~(PGSIZE-1))
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))

// Page table/directory entry flags.
#define PTE_P 0x001 // Present
#define PTE_W 0x002 // Writable
135 #define PTE_U 0x004 // User
#define PTE_PWT 0x008 // Write-Through
#define PTE_PCD 0x010 // Cache-Disable
#define PTE_A 0x020 // Accessed
#define PTE_D 0x040 // Dirty
140 #define PTE_PS 0x080 // Page Size
#define PTE_MBZ 0x180 // Bits must be zero

// Address in page table or page directory entry
145 #define PTE_ADDR(pte) ((uint)(pte) & ~0xFFF)
#define PTE_FLAGS(pte) ((uint)(pte) & 0xFFF)

#ifndef __ASSEMBLER__
typedef uint pte_t;

// Task state segment format
150 struct taskstate {
    uint link; // Old ts selector
    uint esp0; // Stack pointers and segment selectors
    ushort ss0; // after an increase in privilege level
155 #define padding1;
    uint *esp1;
    ushort ss1;
    ushort padding2;
    uint *esp2;
    ushort ss2;
160 #define padding3;
    void *cr3; // Page directory base
    uint *eip; // Saved state from last task switch
    uint eflags;
165 #define padding4;
    uint eax; // More saved state (registers)
    uint ecx;
    uint edx;
    uint ebx;
    uint *esp;
170 #define padding5;
    uint *ebp;
    uint esi;
    uint edi;
    ushort es; // Even more saved state (segment selectors)
    ushort padding4;
175 #define padding6;
    ushort cs;
    ushort padding5;
    ushort ss;
    ushort padding6;
    ushort ds;
180 #define padding7;
    ushort fs;
    ushort padding8;
    ushort gs;
    ushort padding9;
185 #define padding8;
    ushort ldt;
    ushort padding10;
    ushort t; // Trap on task switch
    ushort iomb; // I/O map base address
};

190 // PAGEBREAK: 12
// Gate descriptors for interrupts and traps

```

```

struct gatedesc {
195 #define off_15_0 : 16; // low 16 bits of offset in segment
    uint cs : 16; // code segment selector
    uint args : 5; // # args, 0 for interrupt/trap gates
    uint rsv1 : 3; // reserved (should be zero I guess)
    uint type : 4; // type(STS_TG,IG32,TG32)
200 #define padding1;
    uint s : 1; // must be 0 (system)
    uint dpl : 2; // descriptor (meaning new) privilege level
    uint p : 1; // Present
    uint off_31_16 : 16; // high bits of offset in segment
};

205 // Set up a normal interrupt/trap gate descriptor.
// - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
// - interrupt gate clears FL_IF, trap gate leaves FL_IF alone
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
210 #define dpl: Descriptor Privilege Level -
// the privilege level required for software to invoke
// this interrupt/trap gate explicitly using an int instruction.
#define SETGATE(gate, istrap, sel, off, d) \
{ \
215 (gate).off_15_0 = (uint)(off) & 0xffff; \
(gate).cs = (sel); \
(gate).args = 0; \
(gate).rsv1 = 0; \
(gate).type = (istrap) ? STS_TG32 : STS_IG32; \
220 (gate).s = 0; \
(gate).dpl = (d); \
(gate).p = 1; \
(gate).off_31_16 = (uint)(off) >> 16; \
}

225 #endif

```

```

// Routines to let C code use special x86 instructions.

static inline uchar
inb(ushort port)
5 {
    uchar data;

    asm volatile("in %1,%0" : "=a" (data) : "d" (port));
    return data;
10 }

static inline void
insl(int port, void *addr, int cnt)
15 {
    asm volatile("cld; rep insl" :
        "=D" (addr), "=c" (cnt) :
        "d" (port), "0" (addr), "1" (cnt) :
        "memory", "cc");
20 }

static inline void
outb(ushort port, uchar data)
25 {
    asm volatile("out %0,%1" : "a" (data), "d" (port));
}

static inline void
outw(ushort port, ushort data)
30 {
    asm volatile("out %0,%1" : "a" (data), "d" (port));
}

static inline void
outsl(int port, const void *addr, int cnt)
35 {
    asm volatile("cld; rep outsl" :
        "=S" (addr), "=c" (cnt) :
        "d" (port), "0" (addr), "1" (cnt) :
        "cc");
40 }

static inline void
stosb(void *addr, int data, int cnt)
45 {
    asm volatile("cld; rep stosb" :
        "=D" (addr), "=c" (cnt) :
        "0" (addr), "1" (cnt), "a" (data) :
        "memory", "cc");
50 }

static inline void
stosl(void *addr, int data, int cnt)
55 {
    asm volatile("cld; rep stosl" :
        "=D" (addr), "=c" (cnt) :
        "0" (addr), "1" (cnt), "a" (data) :
        "memory", "cc");
60 }

struct segdesc;

static inline void
lgdt(struct segdesc *p, int size)
{

```

```

65 volatile ushort pd[3];

    pd[0] = size-1;
    pd[1] = (uint)p;
    pd[2] = (uint)p >> 16;
70
    asm volatile("lgdt(%0)" : : "r" (pd));
}

struct gatedesc;

75 static inline void
lidt(struct gatedesc *p, int size)
{
    volatile ushort pd[3];
80
    pd[0] = size-1;
    pd[1] = (uint)p;
    pd[2] = (uint)p >> 16;

    asm volatile("lidt(%0)" : : "r" (pd));
}

static inline void
ltr(ushort sel)
90 {
    asm volatile("ltr %0" : : "r" (sel));
}

static inline uint
readeflags(void)
95 {
    uint eflags;
    asm volatile("pushfl; popl %0" : "=r" (eflags));
    return eflags;
100 }

static inline void
loadgs(ushort v)
105 {
    asm volatile("movw %0, %%gs" : : "r" (v));
}

static inline void
cli(void)
110 {
    asm volatile("cli");
}

static inline void
sti(void)
115 {
    asm volatile("sti");
}

static inline uint
xchg(volatile uint *addr, uint newval)
120 {
    uint result;

125 // The + in "+m" denotes a read-modify-write operand.
    asm volatile("lock; xchgl %0,%1" :
        "+m" (*addr), "=a" (result) :
        "1" (newval) :

```

```

    "cc");
130 return result;
}

static inline uint
rcr2(void)
135 {
    uint val;
    asm volatile("movl %%cr2,%0" : "=r" (val));
    return val;
}

static inline void
lcr3(uint val)
140 {
    asm volatile("movl %0, %%cr3" : : "r" (val));
145 }

//PAGEBREAK: 36
// Layout of the trap frame built on the stack by the
// hardware and by trapasm.S, and passed to trap().
150 struct trapframe {
    // registers as pushed by pusha
    uint edi;
    uint esi;
    uint ebp;
155 uint oesp; // useless & ignored
    uint ebx;
    uint edx;
    uint ecx;
    uint eax;

160 // rest of trap frame
    ushort gs;
    ushort padding1;
    ushort fs;
    ushort padding2;
    ushort es;
    ushort padding3;
    ushort ds;
    ushort padding4;
170 uint trapno;

    // below here defined by x86 hardware
    uint err;
    uint eip;
175 ushort cs;
    ushort padding5;
    uint eflags;

    // below here only when crossing rings, such as from user to kernel
180 uint esp;
    ushort ss;
    ushort padding6;
};

```

```

//
// assembler macros to create x86 segments
//
5 #define SEG_NULLASM \
    .word 0, 0; \
    .byte 0, 0, 0, 0

// The 0xC0 means the limit is in 4096-byte units
// and (for executable segments) 32-bit mode.
10 #define SEG_ASM(type,base,lim) \
    .word (((lim) >> 12) & 0xffff), ((base) & 0xffff); \
    .byte (((base) >> 16) & 0xff), (0x90 | (type)), \
    (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)

15 #define STA_X 0x8 // Executable segment
#define STA_E 0x4 // Expand down (non-executable segments)
#define STA_C 0x4 // Conforming code segment (executable only)
#define STA_W 0x2 // Writeable (non-executable segments)
20 #define STA_R 0x2 // Readable (executable segments)
#define STA_A 0x1 // Accessed

```

```

// Format of an ELF executable file

#define ELF_MAGIC 0x464C457FU // "\x7FELF" in little endian

5 // File header
struct elfhdr {
    uint magic; // must equal ELF_MAGIC
    uchar elf[12];
    ushort type;
    ushort machine;
    uint version;
    uint entry;
    uint phoff;
    uint shoff;
    uint flags;
    ushort ehsize;
    ushort phentsize;
    ushort phnum;
    ushort shentsize;
    ushort shnum;
    ushort shstrndx;
};

// Program section header
25 struct proghdr {
    uint type;
    uint off;
    uint vaddr;
    uint paddr;
    uint filesz;
    uint memsz;
    uint flags;
    uint align;
};

35 // Values for Proghdr type
#define ELF_PROG_LOAD 1

// Flag bits for Proghdr flags
40 #define ELF_PROG_FLAG_EXEC 1
#define ELF_PROG_FLAG_WRITE 2
#define ELF_PROG_FLAG_READ 4

```

```

#include "asm.h"
#include "memlayout.h"
#include "mmu.h"

5 # Start the first CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00.

10 .code16 # Assemble for 16-bit mode
.globl start
start:
    cli # BIOS enabled interrupts; disable

15 # Zero data segment registers DS, ES, and SS.
xorw %ax,%ax # Set %ax to zero
movw %ax,%ds # -> Data Segment
movw %ax,%es # -> Extra Segment
movw %ax,%ss # -> Stack Segment

20 # Physical address line A20 is tied to zero so that the first PCs
# with 2 MB would run software that assumed 1 MB. Undo that.
seta20.1:
    inb $0x64,%al # Wait for not busy
    testb $0x2,%al
    jnz seta20.1

    movb $0xd1,%al # 0xd1 -> port 0x64
    outb %al,$0x64

30 seta20.2:
    inb $0x64,%al # Wait for not busy
    testb $0x2,%al
    jnz seta20.2

35 movb $0xdf,%al # 0xdf -> port 0x60
    outb %al,$0x60

# Switch from real to protected mode. Use a bootstrap GDT that makes
40 # virtual addresses map directly to physical addresses so that the
# effective memory map doesn't change during the transition.
lgdt gdtdesc
movl %cr0, %eax
orl %CR0_PE, %eax
45 movl %eax, %cr0

//PAGEBREAK!
# Complete transition to 32-bit protected mode by using long jmp
# to reload %cs and %ip. The segment descriptors are set up with no
# translation, so that the mapping is still the identity mapping.
50 ljmp $(SEG_KCODE<<3), $start32

.code32 # Tell assembler to generate 32-bit code now.
start32:
# Set up the protected-mode data segment registers
55 movw $(SEG_KDATA<<3), %ax # Our data segment selector
movw %ax, %ds # -> DS: Data Segment
movw %ax, %es # -> ES: Extra Segment
movw %ax, %ss # -> SS: Stack Segment
60 movw $0, %ax # Zero segments not ready for use
movw %ax, %fs # -> FS
movw %ax, %gs # -> GS

# Set up the stack pointer and call into C.

```

```

65 movl $start, %esp
call bootmain

# If bootmain returns (it shouldn't), trigger a Bochs
# breakpoint if running under Bochs, then loop.
70 movw $0x8a00, %ax # 0x8a00 -> port 0x8a00
movw %ax, %dx
outw %ax, %dx
movw $0x8ae0, %ax # 0x8ae0 -> port 0x8a00
outw %ax, %dx

75 spin:
    jmp spin

# Bootstrap GDT
.p2align 2 # force 4 byte alignment
80 gdt:
    SEG_NULLASM # null seg
    SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff) # code seg
    SEG_ASM(STA_W, 0x0, 0xffffffff) # data seg

85 gdtdesc:
    .word (gdt - gdt - 1) # sizeof(gdt) - 1
    .long gdt # address gdt

```

```

// Boot loader.
//
// Part of the boot sector, along with bootasm.S, which calls bootmain().
// bootasm.S has put the processor into protected 32-bit mode.
5 // bootmain() loads an ELF kernel image from the disk starting at
// sector 1 and then jumps to the kernel entry routine.

#include "types.h"
#include "elf.h"
10 #include "x86.h"
#include "memlayout.h"

#define SECTSIZE 512

15 void readseg(uchar*, uint, uint);

void
bootmain(void)
{
20     struct elfhdr *elf;
    struct proghdr *ph, *eph;
    void (*entry)(void);
    uchar *pa;

25     elf = (struct elfhdr*)0x10000; // scratch space

    // Read 1st page off disk
    readseg((uchar*)elf, 4096, 0);

30     // Is this an ELF executable?
    if(elf->magic != ELF_MAGIC)
        return; // let bootasm.S handle error

    // Load each program segment (ignores ph flags).
    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)
40             stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
    }

    // Call the entry point from the ELF header.
    // Does not return!
    entry = (void*)(void)(elf->entry);
    entry();

50 void
waitdisk(void)
{
    // Wait for disk ready.
    while((inb(0x1F7) & 0xC0) != 0x40)
65         ;

    // Read a single sector at offset into dst.
    void
    readsect(void *dst, uint offset)
    {
        // Issue command.
        waitdisk();
        outb(0x1F2, 1); // count = 1
    }
}

```

```

65  outb(0x1F3, offset);
    outb(0x1F4, offset >> 8);
    outb(0x1F5, offset >> 16);
    outb(0x1F6, (offset >> 24) | 0xE0);
    outb(0x1F7, 0x20); // cmd 0x20 - read sectors
70
    // Read data.
    waitdisk();
    insl(0x1F0, dst, SECTSIZE/4);
75
    // Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
    // Might copy more than asked.
    void
    readseg(uchar* pa, uint count, uint offset)
80  {
    uchar* epa;

    epa = pa + count;

85    // Round down to sector boundary.
    pa -= offset % SECTSIZE;

    // Translate from bytes to sectors; kernel starts at sector 1.
    offset = (offset / SECTSIZE) + 1;
90
    // If this is too slow, we could read lots of sectors at a time.
    // We'd write more to memory than asked, but it doesn't matter --
    // we load in increasing order.
    for(; pa < epa; pa += SECTSIZE, offset++)
85    readsect(pa, offset);
}

```

```

# Multiboot header, for multiboot boot loaders like GNU Grub.
# http://www.gnu.org/software/grub/manual/multiboot/multiboot.html
#
# Using GRUB 2, you can boot xv6 from a file stored in a
5 # Linux file system by copying kernel or kernelmemfs to /boot
# and then adding this menu entry:
#
# menuentry "xv6" {
#   insmod ext2
10 #   set root="(hd0,msdos1)"
#   set kernel="/boot/kernel"
#   echo "Loading ${kernel}..."
#   multiboot ${kernel} ${kernel}
#   boot
15 # }

#include "asm.h"
#include "memlayout.h"
#include "mmu.h"
20 #include "param.h"

# Multiboot header. Data to direct multiboot loader.
.p2align 2
.text
25 .globl multiboot_header
multiboot_header:
#define magic 0x1badb002
#define flags 0
    .long magic
    .long flags
30    .long (-magic-flags)

# By convention, the _start symbol specifies the ELF entry point.
# Since we haven't set up virtual memory yet, our entry point is
35 # the physical address of 'entry'.
.globl _start
_start = V2P_WO(entry)

# Entering xv6 on boot processor, with paging off.
40 .globl entry
entry:
# Turn on page size extension for 4Mbyte pages
movl    %cr4, %eax
orl    $(CR4_PSE), %eax
45 movl    %eax, %cr4
# Set page directory
movl    $(V2P_WO(entrypgdir)), %eax
movl    %eax, %cr3
# Turn on paging.
50 movl    %cr0, %eax
orl    $(CR0_PG|CR0_WP), %eax
movl    %eax, %cr0

# Set up the stack pointer.
55 movl    $(stack + KSTACKSIZE), %esp

# Jump to main(), and switch to executing at
# high addresses. The indirect call is needed because
# the assembler produces a PC-relative instruction
# for a direct jump.
60 mov    $main, %eax
jmp    *%eax

.comm stack, KSTACKSIZE

```

```

#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
5 #include "mmu.h"
#include "proc.h"
#include "x86.h"

static void startothers(void);
static void mpmain(void) __attribute__((noreturn));
extern pde_t *kpgdir;
extern char end[]; // first address after kernel loaded from ELF file
10
// Bootstrap processor starts running C code here.
// Allocate a real stack and switch to it, first
// doing some setup required for memory allocator to work.
int
main(void)
20 {
    kinit1(end, P2V(4*1024*1024)); // phys page allocator
    kvmalloc(); // kernel page table
    mpinit(); // collect info about this machine
    lapicinit();
    seginit(); // set up segments
25    printf("cpu%d: starting xv6\n", cpu->id);
    picinit(); // interrupt controller
    ioapicinit(); // another interrupt controller
    consoleinit(); // I/O devices & their interrupts
    uartinit(); // serial port
30    pinit(); // process table
    twinit(); // trap vectors
    binit(); // buffer cache
    fileinit(); // file table
    iinit(); // inode cache
35    ideinit(); // disk
    if(!ismp)
        timerinit(); // uniprocessor timer
    startothers(); // start other processors
    kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
40    userinit(); // first user process
    // Finish setting up this processor in mpmain.
    mpmain();
}

45 // Other CPUs jump here from entryother.S.
static void
mpenter(void)
{
    switchkvm();
50    seginit();
    lapicinit();
    mpmain();
}

55 // Common CPU setup code.
static void
mpmain(void)
{
    printf("cpu%d: starting\n", cpu->id);
60    idtinit(); // load idt register
    xchg(&cpu->started, 1); // tell startothers() we're up
    scheduler(); // start running processes
}

```

```

65 pde_t entrypgdir[]; // For entry.S

// Start the non-boot (AP) processors.
static void
startothers(void)
70 {
    extern uchar _binary_entryother_start[], _binary_entryother_size[];
    uchar *code;
    struct cpu *c;
    char *stack;

75
    // Write entry code to unused memory at 0x7000.
    // The linker has placed the image of entryother.S in
    // _binary_entryother_start.
    code = p2v(0x7000);
    memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);
80
    for(c = cpus; c < cpus+ncpu; c++){
        if(c == cpus+cpunum()) // We've started already.
            continue;
85
        // Tell entryother.S what stack to use, where to enter, and what
        // pgdir to use. We cannot use kpgdir yet, because the AP processor
        // is running in low memory, so we use entrypgdir for the APs too.
        stack = kalloc();
        *(void**) (code-4) = stack + KSTACKSIZE;
        *(void**) (code-8) = mpenter;
        *(int**) (code-12) = (void *) v2p(entrypgdir);
90
        lapicstartap(c->id, v2p(code));
95
        // wait for cpu to finish mpmain()
        while(c->started == 0)
            ;
100 }

// Boot page table used in entry.S and entryother.S.
// Page directories (and page tables), must start on a page boundary,
// hence the " _aligned_" attribute.
105 // Use PTE_PS in page directory entry to enable 4Mbyte pages.
__attribute__((aligned(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
    // Map VA's [0, 4MB) to PA's [0, 4MB)
    [0] = (0) | PTE_P | PTE_W | PTE_PS,
110    // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
    [KERNBASE >> PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};

//PAGEBREAK!
115 // Blank page.

```

```

#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
seginitt(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_USER, but the CPU forbids
    // an interrupt from CPL=0 to DPL=3.
    c = &cpu[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;

    // There is one page table per process, plus one that's used when
    // a CPU is not running any process (kpgdir). The kernel uses the
    // current process's page table during system calls and interrupts;
    // page protection bits prevent user code from using the kernel's
    // mappings.
    // setupkvm() and exec() set up every page table like this:
    // 0..KERNBASE: user memory (text+data+stack+heap), mapped to
    // phys memory allocated by the kernel
    // KERNBASE..KERNBASE+EXTMEM: mapped to 0..EXTMEM (for I/O space)
    // KERNBASE+EXTMEM..data: mapped to EXTMEM..V2P(data)
    // for the kernel's instructions and r/o data
    // data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
    // rw data + free physical memory
    // 0xfe000000..0: mapped direct (devices such as ioapic)
    // The kernel allocates physical memory for its heap and for user memory
    // between V2P(end) and the end of physical memory (PHYSTOP)
    // (directly addressable from end.P2V(PHYSTOP)).

    // This table defines the kernel's mappings, which are present in
    // every process's page table.
    static struct kmap {
        void *virt;
        uint phys_start;
        int perm;
    } kmap[] = {
        { (void*)KERNBASE, 0, EXTMEM, PTE_W}, // I/O space
        { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kern text+rodata
        { (void*)data, V2P(data), PHYSTOP, PTE_W}, // kern data+memory
        { (void*)DEVSPACE, DEVSPACE, 0, PTE_W}, // more devices
    };

    // Set up kernel part of a page table.
    pde_t *

```

```

setupkvm(void)
{
    pde_t *pgdir;
    struct kmap *k;

    if((pgdir = (pde_t*)kalloc()) == 0)
        return 0;
    memset(pgdir, 0, PGSIZE);
    if (p2v(PHYSTOP) > (void*)DEVSPACE)
        panic("PHYSTOP too high");
    for(k = kmap; k < &kmap[NELEM(kmap)]; k++){
        if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
            (uint)k->phys_start, k->perm) < 0)
            return 0;
        return pgdir;
    }

    // Allocate one page table for the machine for the kernel address
    // space for scheduler processes.
    void
    kvmalloc(void)
    {
        kpgdir = setupkvm();
        switchkvm();
    }

    // Switch h/w page table register to the kernel-only page table,
    // for when no process is running.
    void
    switchkvm(void)
    {
        lcr3(v2p(kpgdir)); // switch to the kernel page table

        // Switch TSS and h/w page table to correspond to process p.
        void
        switchvm(struct proc *p)
        {
            pushcli();
            cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
            cpu->gdt[SEG_TSS].s = 0;
            cpu->ts.ss0 = SEG_KDATA << 3;
            cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
            ltr(SEG_TSS << 3);
            if(p->pgdir == 0)
                panic("switchvm: no pgdir");
            lcr3(v2p(p->pgdir)); // switch to new address space
            popcli();
        }

        // Load the initcode into address 0 of pgdir.
        // sz must be less than a page.
        void
        initvm(pde_t *pgdir, char *init, uint sz)
        {
            char *mem;

            if(sz >= PGSIZE)
                panic("initvm: more than a page");
            mem = kalloc();
            memset(mem, 0, PGSIZE);
            mappages(pgdir, 0, PGSIZE, v2p(mem), PTE_W|PTE_U);
            memmove(mem, init, sz);
        }
    }

```

```

// Load a program segment into pgdir. addr must be page-aligned
// and the pages from addr to addr+sz must already be mapped.
int
loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
    uint i, pa, n;
    pte_t *pte;

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

    // Allocate page tables and physical memory to grow process from oldsz to
    // newsz, which need not be page aligned. Returns new size or 0 on error.
    int
    allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
    {
        char *mem;
        uint a;

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

        a = PGROUNDUP(oldsz);
        for(; a < newsz; a += PGSIZE){
            mem = kalloc();
            if(mem == 0){
                cprintf("allocuvm out of memory\n");
                deallocuvm(pgdir, newsz, oldsz);
                return 0;
            }
            memset(mem, 0, PGSIZE);
            mappages(pgdir, (char*)a, PGSIZE, v2p(mem), PTE_W|PTE_U);
        }
        return newsz;
    }

    // Deallocate user pages to bring the process size from oldsz to
    // newsz. oldsz and newsz need not be page-aligned, nor does newsz
    // need to be less than oldsz. oldsz can be larger than the actual
    // process size. Returns the new process size.
    int
    deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
    {
        pte_t *pte;
        uint a, pa;

        if(newsz >= oldsz)
            return oldsz;
    }

```

```

a = PGROUNDUP(newsz);
for(; a < olds; a += PGSIZE){
260 pte = walkpgdir(pgdir, (char*)a, 0);
    if(!pte)
        a += (NPENTRIES - 1) * PGSIZE;
    else if((*pte & PTE_P) != 0){
265         pa = PTE_ADDR(*pte);
        if(pa == 0)
            panic("kfree");
        char *v = p2v(pa);
        kfree(v);
        *pte = 0;
270     }
}
return newsz;
}

275 // Free a page table and all the physical memory pages
// in the user part.
void
freevm(pde_t *pgdir)
{
280     uint i;

    if(pgdir == 0)
        panic("freevm: no pgdir");
    dealloccvm(pgdir, KERNBASE, 0);
285     for(i = 0; i < NPDEENTRIES; i++){
        if(pgdir[i] & PTE_P){
            char *v = p2v(PTE_ADDR(pgdir[i]));
            kfree(v);
        }
290     }
    kfree((char*)pgdir);
}

// Clear PTE_U on a page. Used to create an inaccessible
295 // page beneath the user stack.
void
clearpte(pde_t *pgdir, char *uva)
{
    pte_t *pte;
300
    pte = walkpgdir(pgdir, uva, 0);
    if(pte == 0)
        panic("clearpte");
    *pte &= ~PTE_U;
305 }

// Given a parent process's page table, create a copy
// of it for a child.
pde_t*
310 copyvm(pde_t *pgdir, uint sz)
{
    pde_t *d;
    pte_t *pte;
    uint pa, i, flags;
315     char *mem;

    if((d = setupkvm()) == 0)
        return 0;
    for(i = 0; i < sz; i += PGSIZE){
320         if(pte = walkpgdir(pgdir, (void *) i, 0)) == 0)

```

```

        panic("copyvm: pte should exist");
        if(!(*pte & PTE_P))
            panic("copyvm: page not present");
        pa = PTE_ADDR(*pte);
325         flags = PTE_FLAGS(*pte);
        if((mem = kalloc()) == 0)
            goto bad;
        memmove(mem, (char*)p2v(pa), PGSIZE);
        if(mappages(d, (void*)i, PGSIZE, v2p(mem), flags) < 0)
330             goto bad;
    }
    return d;
}

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

//PAGEBREAK!
// Map user virtual address to kernel address.
340 char*
uva2ka(pde_t *pgdir, char *uva)
{
    pte_t *pte;
345
    pte = walkpgdir(pgdir, uva, 0);
    if((*pte & PTE_P) == 0)
        return 0;
    if((*pte & PTE_U) == 0)
350         return 0;
    return (char*)p2v(PTE_ADDR(*pte));
}

// Copy len bytes from p to user address va in page table pgdir.
355 // Most useful when pgdir is not the current page table.
// uva2ka ensures this only works for PTE_U pages.
int
copyout(pde_t *pgdir, uint va, void *p, uint len)
{
360     char *buf, *pa0;
    uint n, va0;

    buf = (char*)p;
    while(len > 0){
365         va0 = (uint)PGROUNDDOWN(va);
        pa0 = uva2ka(pgdir, (char*)va0);
        if(pa0 == 0)
            return -1;
        n = PGSIZE - (va - va0);
370         if(n > len)
            n = len;
        memmove(pa0 + (va - va0), buf, n);
        len -= n;
        buf += n;
375         va = va0 + PGSIZE;
    }
    return 0;
}

```

traps.h

```

// x86 trap and interrupt constants.

// Processor-defined:
#define T_DIVIDE      0 // divide error
5 #define T_DEBUG     1 // debug exception
#define T_NMI        2 // non-maskable interrupt
#define T_BRKPT     3 // breakpoint
#define T_OFLOW     4 // overflow
10 #define T_BOUND    5 // bounds check
#define T_ILLOP    6 // illegal opcode
#define T_DEVICE    7 // device not available
#define T_DBLFLT   8 // double fault
// #define T_COPROC  9 // reserved (not used since 486)
15 #define T_TSS     10 // invalid task switch segment
#define T_SEGNP   11 // segment not present
#define T_STACK   12 // stack exception
#define T_GPFLT   13 // general protection fault
#define T_PGFLT   14 // page fault
20 // #define T_RES    15 // reserved
#define T_FPERR   16 // floating point error
#define T_ALIGN  17 // alignment check
#define T_MCHK    18 // machine check
#define T_SIMDERR 19 // SIMD floating point error
25
// These are arbitrarily chosen, but with care not to overlap
// processor defined exceptions or interrupt vectors.
#define T_SYSCALL  64 // system call
#define T_DEFAULT  500 // catchall
30
#define T_IRQ0    32 // IRQ 0 corresponds to int T_IRQ

#define IRQ_TIMER  0
#define IRQ_KBD    1
#define IRQ_COM1   4
35 #define IRQ_IDE   14
#define IRQ_ERROR  19
#define IRQ_SPURIOUS 31

```

vectors.pl

```

#!/usr/bin/perl -w

# Generate vectors.S, the trap/interrupt entry points.
# There has to be one entry point per interrupt number
5 # since otherwise there's no way for trap() to discover
# the interrupt number.

print "# generated by vectors.pl - do not edit!\n";
print "# handlers\n";
10 print ".globl alltraps\n";
for(my $i = 0; $i < 256; $i++){
    print ".globl vector${i}\n";
    print ".vector${i}\n";
    if(!($i == 8 || ($i >= 10 && $i <= 14) || $i == 17)){
15         print ".pushl \${0}\n";
    }
    print ".pushl \${$i}\n";
    print ".jmp alltraps\n";
}

20
print "\n# vector table\n";
print ".data\n";
print ".globl vectors\n";
print "vectors:\n";
25 for(my $i = 0; $i < 256; $i++){
    print ".long vector${i}\n";
}

# sample output:
30 # handlers
# .globl alltraps
# .globl vector0
# vector0:
#     pushl $0
35 #     pushl $0
#     jmp alltraps
# ...
#
# vector table
40 # .data
# .globl vectors
# vectors:
#     .long vector0
#     .long vector1
45 #     .long vector2
# ...

```



```

#include "mmu.h"

# vectors.S sends all traps here.
.globl alltraps
5 alltraps:
# Build trap frame.
pushl %ds
pushl %es
pushl %fs
10 pushl %gs
pushal

# Set up data and per-cpu segments.
movw $(SEG_KDATA<<3), %ax
15 movw %ax, %ds
movw %ax, %es
movw $(SEG_KCPU<<3), %ax
movw %ax, %fs
movw %ax, %gs

20 # Call trap(tf), where tf=%esp
pushl %esp
call trap
addl $4, %esp

25 # Return falls through to trapret...
.globl trapret
trapret:
popal
30 popl %gs
popl %fs
popl %es
popl %ds
addl $0x8, %esp # trapno and errcode
35 iret

```

```

#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
5 #include "mmu.h"
#include "proc.h"
#include "x86.h"
#include "traps.h"
#include "spinlock.h"

10 // 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;
15 uint ticks;

void
tvinit(void)
{
20 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);

25 initlock(&tickslock, "time");
}

void
idtinit(void)
{
30 lidt(idt, sizeof(idt));
}

//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
40 if(tf->trapno == T_SYSCALL){
if(proc->killed)
exit();
proc->tf = tf;
syscall();
if(proc->killed)
45 exit();
return;
}

switch(tf->trapno){
50 case T_IRQ0 + IRQ_TIMER:
if(cpu->id == 0){
acquire(&tickslock);
ticks++;
wakeup(&ticks);
55 release(&tickslock);
}
lapiceoi();
break;
case T_IRQ0 + IRQ_IDE:
ideintr();
lapiceoi();
60 break;
case T_IRQ0 + IRQ_IDE+1:
// Bochs generates spurious IDE1 interrupts.

```

```

65 break;
case T_IRQ0 + IRQ_KBD:
kbdintr();
lapiceoi();
break;
70 case T_IRQ0 + IRQ_COM1:
uartintr();
lapiceoi();
break;
case T_IRQ0 + 7:
75 case T_IRQ0 + IRQ_SPURIOUS:
cprintf("cpu%d: spurious interrupt at %x:%x\n",
cpu->id, tf->cs, tf->eip);
lapiceoi();
break;

80 //PAGEBREAK: 13
default:
if(proc == 0 || (tf->cs&3) == 0){
// In kernel, it must be our mistake.
85 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.
90 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;
95 }

// 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.)
100 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.
105 if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
yield();

// Check if the process has been killed since we yielded
110 if(proc && proc->killed && (tf->cs&3) == DPL_USER)
exit();
}

```

```

// System call numbers
#define SYS_fork 1
#define SYS_exit 2
#define SYS_wait 3
5 #define SYS_pipe 4
#define SYS_read 5
#define SYS_kill 6
#define SYS_exec 7
#define SYS_fstat 8
10 #define SYS_chdir 9
#define SYS_dup 10
#define SYS_getpid 11
#define SYS_sbrk 12
#define SYS_sleep 13
15 #define SYS_uptime 14
#define SYS_open 15
#define SYS_write 16
#define SYS_mknod 17
#define SYS_unlink 18
20 #define SYS_link 19
#define SYS_mkdir 20
#define SYS_close 21

```

```

#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
5 #include "mmu.h"
#include "proc.h"
#include "x86.h"
#include "syscall.h"

10 // 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.

15 // Fetch the int at addr from the current process.
int
fetchint(uint addr, int *ip)
{
20     if(addr >= proc->sz || addr+4 > proc->sz)
        return -1;
    *ip = *(int*)(addr);
    return 0;
}

25 // 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)
{
30     char *s, *ep;

    if(addr >= proc->sz)
35         return -1;
    *pp = (char*)addr;
    ep = (char*)proc->sz;
    for(s = *pp; s < ep; s++)
        if(*s == 0)
40             return s - *pp;
    return -1;
}

// Fetch the nth 32-bit system call argument.
45 int
argint(int n, int *ip)
{
    return fetchint(proc->tf->esp + 4 + 4*n, ip);
}

50 // 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
55 argptr(int n, char **pp, int size)
{
    int i;

    if(argint(n, &i) < 0)
60         return -1;
    if((uint)i >= proc->sz || (uint)i+size > proc->sz)
        return -1;
    *pp = (char*)i;
    return 0;
}

```

```

int num;

130 num = proc->tf->eax;
if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    proc->tf->eax = syscalls[num]();
} else {
135     cprintf("%d %s: unknown sys call %d\n",
        proc->pid, proc->name, num);
    proc->tf->eax = -1;
}
}

```

```

65 }

// 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
70 // between this check and being used by the kernel.)
int
argstr(int n, char **pp)
{
    int addr;
75     if(argint(n, &addr) < 0)
        return -1;
    return fetchstr(addr, pp);
}

80 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);
85 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);
90 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);
95 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);
100 extern int sys_uptime(void);

static int (*syscalls[])(void) = {
[SYS_fork] sys_fork,
[SYS_exit] sys_exit,
105 [SYS_wait] sys_wait,
[SYS_pipe] sys_pipe,
[SYS_read] sys_read,
[SYS_kill] sys_kill,
[SYS_exec] sys_exec,
110 [SYS_fstat] sys_fstat,
[SYS_chdir] sys_chdir,
[SYS_dup] sys_dup,
[SYS_getpid] sys_getpid,
[SYS_sbrk] sys_sbrk,
115 [SYS_sleep] sys_sleep,
[SYS_uptime] sys_uptime,
[SYS_open] sys_open,
[SYS_write] sys_write,
[SYS_mknod] sys_mknod,
[SYS_unlink] sys_unlink,
120 [SYS_link] sys_link,
[SYS_mkdir] sys_mkdir,
[SYS_close] sys_close,
};

125 void
syscall(void)
{

```

```

#include "types.h"
#include "x86.h"
#include "defs.h"
#include "param.h"
5 #include "memlayout.h"
#include "mmu.h"
#include "proc.h"

int
10 sys_fork(void)
{
    return fork();
}

15 int
sys_exit(void)
{
    exit();
    return 0; // not reached
20 }

int
sys_wait(void)
{
25     return wait();
}

int
sys_kill(void)
30 {
    int pid;

    if(argint(0, &pid) < 0)
35         return -1;
    return kill(pid);
}

int
sys_getpid(void)
40 {
    return proc->pid;
}

int
45 sys_sbrk(void)
{
    int addr;
    int n;

50     if(argint(0, &n) < 0)
        return -1;
    addr = proc->sz;
    if(growproc(n) < 0)
        return -1;
    return addr;
}

int
55 sys_sleep(void)
{
    int n;
    uint ticks0;

60     if(argint(0, &n) < 0)

```

```

65     return -1;
    acquire(&tickslock);
    ticks0 = ticks;
    while(ticks - ticks0 < n){
    if(proc->killed){
70         release(&tickslock);
        return -1;
    }
    sleep(&ticks, &tickslock);
    }
    release(&tickslock);
75     return 0;
    }

// return how many clock tick interrupts have occurred
80 // since start.
int
sys_uptime(void)
{
    uint xticks;
85     acquire(&tickslock);
    xticks = ticks;
    release(&tickslock);
    return xticks;
90 }

```

```

#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
5  #include "mmu.h"
#include "x86.h"
#include "proc.h"
#include "spinlock.h"

10 struct {
    struct spinlock lock;
    struct proc proc[NPROC];
} ptable;

15 static struct proc *initproc;

int nextpid = 1;
extern void forkret(void);
extern void trapret(void);

20 static void wakeup1(void *chan);

void
pinit(void)
25 {
    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*
35 allocproc(void)
{
    struct proc *p;
    char *sp;

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

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

        // Allocate kernel stack.
        if((p->kstack = kalloc()) == 0){
            p->state = UNUSED;
55     return 0;
        }
        sp = p->kstack + KSTACKSIZE;

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

        // Set up new context to start executing at forkret,
        // which returns to trapret.

```

```

65     sp -= 4;
    *(uint*)sp = (uint)trapret;

    sp -= sizeof *p->context;
    p->context = (struct context*)sp;
70     memset(p->context, 0, sizeof *p->context);
    p->context->eip = (uint)forkret;

    return p;
}

75 //PAGEBREAK: 32
// Set up first user process.
void
userinit(void)
80 {
    struct proc *p;
    extern char _binary_initcode_start[], _binary_initcode_size[];

    p = allocproc();
    initproc = p;
85     if((p->pgdir = setupkvm()) == 0)
        panic("userinit: out of memory?");
    inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
    p->sz = PGSIZE;
90     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;
95     p->tf->eflags = FL_IF;
    p->tf->esp = PGSIZE;
    p->tf->eip = 0; // beginning of initcode.S

    safestrcpy(p->name, "initcode", sizeof(p->name));
100     p->cwd = namei("/");

    p->state = RUNNABLE;
}

105 // Grow current process's memory by n bytes.
// Return 0 on success, -1 on failure.
int
growproc(int n)
{
110     uint sz;

    sz = proc->sz;
    if(n > 0){
        if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
115     return -1;
    } else if(n < 0){
        if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
            return -1;
    }
    proc->sz = sz;
    switchuvm(proc);
120     return 0;
}

125 // 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)
130 {
    int i, pid;
    struct proc *np;

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

    // Copy process state from p.
    if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
140         kfree(np->kstack);
        np->kstack = 0;
        np->state = UNUSED;
        return -1;
    }
    np->sz = proc->sz;
    np->parent = proc;
    *np->tf = *proc->tf;

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

    for(i = 0; i < NOFILE; i++)
        if(proc->ofile[i])
150         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));
160     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)
{
170     struct proc *p;
    int fd;

    if(proc == initproc)
        panic("init exiting");

175     // Close all open files.
    for(fd = 0; fd < NOFILE; fd++){
        if(proc->ofile[fd]){
            fileclose(proc->ofile[fd]);
            proc->ofile[fd] = 0;
180         }
    }

    iput(proc->cwd);
    proc->cwd = 0;

185     acquire(&ptable.lock);

    // Parent might be sleeping in wait().
    wakeup1(proc->parent);

190     // Pass abandoned children to init.
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

```

```

195     if(p->parent == proc){
        p->parent = initproc;
        if(p->state == ZOMBIE)
            wakeupl(initproc);
    }
}

200 // Jump into the scheduler, never to return.
proc->state = ZOMBIE;
sched();
panic("zombie exit");
}

205 // Wait for a child process to exit and return its pid.
// Return -1 if this process has no children.
int
wait(void)
210 {
    struct proc *p;
    int havekids, pid;

    acquire(&ptable.lock);
215     for(;;){
        // Scan through table looking for zombie children.
        havekids = 0;
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->parent != proc)
220                 continue;
            havekids = 1;
            if(p->state == ZOMBIE){
                // Found one.
                pid = p->pid;
225                 kfree(p->kstack);
                p->kstack = 0;
                freevm(p->pgdir);
                p->state = UNUSED;
                p->pid = 0;
                p->parent = 0;
230                 p->name[0] = 0;
                p->killed = 0;
                release(&ptable.lock);
                return pid;
            }
        }

        // No point waiting if we don't have any children.
240         if(!havekids || proc->killed){
            release(&ptable.lock);
            return -1;
        }

        // Wait for children to exit. (See wakeupl call in proc_exit.)
245         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
255     // - eventually that process transfers control
    //   via swtch back to the scheduler.

```

```

void
scheduler(void)
260 {
    struct proc *p;

    for(;;){
        // Enable interrupts on this processor.
        sti();
    }

265     // 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)
270             continue;

        // Switch to chosen process. It is the process's job
        // to release ptable.lock and then reacquire it
        // before jumping back to us.
275         proc = p;
        switchvm(p);
        p->state = RUNNING;
        swtch(&cpu->schedulr, proc->context);
        switchkvm();

280         // Process is done running for now.
        // It should have changed its p->state before coming back.
        proc = 0;
    }
    release(&ptable.lock);
}

290 // Enter scheduler. Must hold only ptable.lock
// and have changed proc->state.
void
sched(void)
295 {
    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_INT)
        panic("sched interruptible");
    intena = cpu->intena;
    swtch(&proc->context, cpu->schedulr);
    cpu->intena = intena;
}

310 // Give up the CPU for one scheduling round.
void
yield(void)
{
    acquire(&ptable.lock); //DOC: yieldlock
315     proc->state = RUNNABLE;
    sched();
    release(&ptable.lock);
}

320 // A fork child's very first scheduling by scheduler()

```

```

// will swtch here. "Return" to user space.
void
forkret(void)
325 {
    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).
}

340 // Atomically release lock and sleep on chan.
// Reacquires lock when awakened.
void
sleep(void *chan, struct spinlock *lk)
345 {
    if(proc == 0)
        panic("sleep");

    if(lk == 0)
        panic("sleep without lk");

350     // 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);
    }

360     // Go to sleep.
    proc->chan = chan;
    proc->state = SLEEPING;
    sched();

    // Tidy up.
    proc->chan = 0;

370     // 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.
380 static void
wakeupl(void *chan)
{
    struct proc *p;

```

```

385     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state == SLEEPING && p->chan == chan)
            p->state = RUNNABLE;
    }

390     // Wake up all processes sleeping on chan.
    void
    wakeup(void *chan)
    {
        acquire(&ptable.lock);
        wakeupl(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)
405 {
    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;
}

420 //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)
425 {
    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];

440     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 = "???" ;
    }
}

```

```

450     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]);
455     }
        cprintf("\n");
    }
}

```

```

# Context switch
# void swtch(struct context **old, struct context *new);
#
5 # Save current register context in old
# and then load register context from new.

.globl swtch
swtch:
10     movl 4(%esp), %eax
        movl 8(%esp), %edx

        # Save old callee-save registers
15     pushl %ebp
        pushl %ebx
        pushl %esi
        pushl %edi

        # Switch stacks
20     movl %esp, (%eax)
        movl %edx, %esp

        # Load new callee-save registers
25     popl %edi
        popl %esi
        popl %ebx
        popl %ebp
        ret

```

```

// Physical memory allocator, intended to allocate
// memory for user processes, kernel stacks, page table pages,
// and pipe buffers. Allocates 4096-byte pages.

5 #include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
10 #include "spinlock.h"

void freerange(void *vstart, void *vend);
extern char end[]; // first address after kernel loaded from ELF file

15 struct run {
    struct run *next;
};

struct {
20     struct spinlock lock;
    int use_lock;
    struct run *freelist;
} kmem;

25 // 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.
30 void
kinit1(void *vstart, void *vend)
{
    initlock(&kmem.lock, "kmem");
    kmem.use_lock = 0;
35     freerange(vstart, vend);
}

void
kinit2(void *vstart, void *vend)
40 {
    freerange(vstart, vend);
    kmem.use_lock = 1;
}

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

//PAGEBREAK: 21
55 // 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
60 kfree(char *v)
{
    struct run *r;

    if((uint)v % PGSIZE || v < end || v2p(v) >= PHYSTOP)

```

```

65     panic("kfree");

    // Fill with junk to catch dangling refs.
    memset(v, 1, PGSIZE);

70     if(kmem.use_lock)
        acquire(&kmem.lock);
    r = (struct run*)v;
    r->next = kmem.freelist;
    kmem.freelist = r;
75     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)
85 {
    struct run *r;

    if(kmem.use_lock)
        acquire(&kmem.lock);
    r = kmem.freelist;
90     if(r)
        kmem.freelist = r->next;
    if(kmem.use_lock)
        release(&kmem.lock);
    return (char*)r;
95 }

```

```

// Mutual exclusion lock.
struct spinlock {
    uint locked; // Is the lock held?

5 // For debugging:
    char *name; // Name of lock.
    struct cpu *cpu; // The cpu holding the lock.
    uint pcs[10]; // The call stack (an array of program counters)
                // that locked the lock.
10 };

```

```

// Mutual exclusion spin locks.

#include "types.h"
#include "defs.h"
5 #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)
15 {
    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
25 acquire(struct spinlock *lk)
{
    pushcli(); // disable interrupts to avoid deadlock.
    if(holding(lk))
        panic("acquire");

30 // The xchg is atomic.
// It also serializes, so that reads after acquire are not
// reordered before it.
while(xchg(&lk->locked, 1) != 0)
35 ;

// Record info about lock acquisition for debugging.
lk->cpu = cpu;
getcallerpcs(&lk, lk->pcs);
40 }

// Release the lock.
void
release(struct spinlock *lk)
45 {
    if(!holding(lk))
        panic("release");

    lk->pcs[0] = 0;
50 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
60 // the above assignments (and after the critical section).
xchg(&lk->locked, 0);

    popcli();
}

```

```

65 // Record the current call stack in pcs[] by following the %ebp chain.
void
getcallerpcs(void *v, uint pcs[])
{
70     uint *ebp;
    int i;

    ebp = (uint*)v - 2;
    for(i = 0; i < 10; i++){
75         if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff)
            break; // saved %eip
        pcs[i] = ebp[1]; // saved %ebp
        ebp = (uint*)ebp[0]; // saved %ebp
80     }
    for(; i < 10; i++)
        pcs[i] = 0;
}

// Check whether this cpu is holding the lock.
int
85 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.

95 void
pushcli(void)
{
    int eflags;

100     eflags = readeflags();
    cli();
    if(cpu->ncli++ == 0)
        cpu->intena = eflags & FL_IF;
105 }

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

```

```

# Initial process execs /init.

#include "syscall.h"
#include "traps.h"
5

# exec(init, argv)
.globl start
start:
10     pushl %argv
    pushl $init
    pushl $0 // where caller pc would be
    movl $$SYS_exec, %eax
    int $T_SYSCALL
15

# for(;;) exit();
exit:
    movl $$SYS_exit, %eax
    int $T_SYSCALL
20     jmp exit

# char init[] = "/init\0";
init:
    .string "/init0"
25

# char *argv[] = { init, 0 };
.p2align 2
argv:
    .long init
30     .long 0

```

```

#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
5   .globl name; \
   name: \
   movl $SYS_ ## name, %eax; \
   int $T_SYSCALL; \
   ret

10  SYSCALL(fork)
   SYSCALL(exit)
   SYSCALL(wait)
   SYSCALL(pipe)
15  SYSCALL(read)
   SYSCALL(write)
   SYSCALL(close)
   SYSCALL(kill)
   SYSCALL(exec)
20  SYSCALL(open)
   SYSCALL(mknod)
   SYSCALL(unlink)
   SYSCALL(fstat)
   SYSCALL(link)
25  SYSCALL(mkdir)
   SYSCALL(chdir)
   SYSCALL(dup)
   SYSCALL(getpid)
   SYSCALL(sork)
30  SYSCALL(sleep)
   SYSCALL(uptime)

```

```

// init: The initial user-level program

#include "types.h"
#include "stat.h"
5  #include "user.h"
#include "fcntl.h"

char *argv[] = { "sh", 0 };

10 int
main(void)
{
   int pid, wpid;

15   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();
25      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)
35          printf(1, "zombie\n");
   }
}

```

```

// Shell.

#include "types.h"
#include "user.h"
5  #include "fcntl.h"

// Parsed command representation
#define EXEC 1
#define REDIR 2
10 #define PIPE 3
#define LIST 4
#define BACK 5

#define MAXARGS 10

15 struct cmd {
   int type;
};

20 struct execcmd {
   int type;
   char *argv[MAXARGS];
   char *eargv[MAXARGS];
};

25 struct redircmd {
   int type;
   struct cmd *cmd;
   char *file;
30  char *efile;
   int mode;
   int fd;
};

35 struct pipecmd {
   int type;
   struct cmd *left;
   struct cmd *right;
};

40 struct listcmd {
   int type;
   struct cmd *left;
   struct cmd *right;
45 };

struct backcmd {
   int type;
   struct cmd *cmd;
50 };

int fork1(void); // Fork but panics on failure.
void panic(char*);
struct cmd *parsecmd(char*);

55 // Execute cmd. Never returns.
void
runcmd(struct cmd *cmd)
{
60   int p[2];
   struct backcmd *bcmd;
   struct execcmd *ecmd;
   struct listcmd *lcmd;
   struct pipecmd *pcmd;

```

```

65  struct redircmd *rcmd;

   if(cmd == 0)
       exit();

70  switch(cmd->type){
   default:
       panic("runcmd");

   case EXEC:
75     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]);
80     break;

   case REDIR:
       rcmd = (struct redircmd*)cmd;
       close(rcmd->fd);
85     if(open(rcmd->file, rcmd->mode) < 0){
           printf(2, "open %s failed\n", rcmd->file);
           exit();
       }
       runcmd(rcmd->cmd);
90     break;

   case LIST:
       lcmd = (struct listcmd*)cmd;
       if(fork1() == 0)
95         runcmd(lcmd->left);
       wait();
       runcmd(lcmd->right);
       break;

100  case PIPE:
       pcmd = (struct pipecmd*)cmd;
       if(pipe(p) < 0)
           panic("pipe");
       if(fork1() == 0){
105         close(1);
           dup(p[1]);
           close(p[0]);
           close(p[1]);
           runcmd(pcmd->left);

110        }
       if(fork1() == 0){
           close(0);
           dup(p[0]);
           close(p[0]);
           close(p[1]);
115         runcmd(pcmd->right);
       }
       close(p[0]);
       close(p[1]);
       wait();
       break;

120  case BACK:
       bcmd = (struct backcmd*)cmd;
       if(fork1() == 0)
125         runcmd(bcmd->cmd);
       break;

```

```

    }
    exit();
}

int
getcmd(char *buf, int nbuf)
135 {
    printf(2, "$ ");
    memset(buf, 0, nbuf);
    gets(buf, nbuf);
    if(buf[0] == 0) // EOF
140     return -1;
    return 0;
}

int
145 main(void)
{
    static char buf[100];
    int fd;

    // Assumes three file descriptors open.
    while((fd = open("console", O_RDWR)) >= 0){
        if(fd >= 3){
            close(fd);
            break;
155     }
    }

    // Read and run input commands.
    while(getcmd(buf, sizeof(buf)) >= 0){
160     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
        // Clumsy but will have to do for now.
        // Chdir has no effect on the parent if run in the child.
        buf[strlen(buf)-1] = 0; // chop \n
        if(chdir(buf+3) < 0)
165             printf(2, "cannot cd %s\n", buf+3);
        continue;
    }
    if(fork1() == 0)
        runcmd(parsecmd(buf));
170     wait();
    }
    exit();
}

void
panic(char *s)
175 {
    printf(2, "%s\n", s);
    exit();
}

int
fork1(void)
185 {
    int pid;

    pid = fork();
    if(pid == -1)
        panic("fork");
190     return pid;
}

```

```

//PAGEBREAK!
// Constructors

195 struct cmd*
execcmd(void)
{
    struct execcmd *cmd;

200     cmd = malloc(sizeof(*cmd));
    memset(cmd, 0, sizeof(*cmd));
    cmd->type = EXEC;
    return (struct cmd*)cmd;
}

205 struct cmd*
redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
{
    struct redircmd *cmd;

210     cmd = malloc(sizeof(*cmd));
    memset(cmd, 0, sizeof(*cmd));
    cmd->type = REDIR;
    cmd->cmd = subcmd;
    cmd->file = file;
    cmd->efile = efile;
    cmd->mode = mode;
    cmd->fd = fd;
220     return (struct cmd*)cmd;
}

struct cmd*
pipecmd(struct cmd *left, struct cmd *right)
225 {
    struct pipecmd *cmd;

    cmd = malloc(sizeof(*cmd));
    memset(cmd, 0, sizeof(*cmd));
    cmd->type = PIPE;
    cmd->left = left;
    cmd->right = right;
    return (struct cmd*)cmd;
}

235 struct cmd*
listcmd(struct cmd *left, struct cmd *right)
{
    struct listcmd *cmd;

240     cmd = malloc(sizeof(*cmd));
    memset(cmd, 0, sizeof(*cmd));
    cmd->type = LIST;
    cmd->left = left;
    cmd->right = right;
    return (struct cmd*)cmd;
}

245 struct cmd*
backcmd(struct cmd *subcmd)
{
    struct backcmd *cmd;

250     cmd = malloc(sizeof(*cmd));
    memset(cmd, 0, sizeof(*cmd));
    cmd->type = BACK;
}

```

```

    cmd->cmd = subcmd;
    return (struct cmd*)cmd;
}

260 //PAGEBREAK!
// Parsing

char whitespace[] = " \t\n\r";
char symbols[] = "<>|:.,;&";

265 int
gettoken(char **ps, char *es, char **q, char **eq)
{
    char *s;
    int ret;

270     s = *ps;
    while(s < es && strchr(whitespace, *s))
        s++;
    if(*q = s;
    ret = *s;
    switch(*s){
280     case 0:
        break;
    case '|':
    case ':':
    case '.':
    case '&':
285     case '<':
        s++;
        break;
    case '>':
290     if(*s == '>'){
        ret = '+';
        s++;
    }
    break;
295     default:
        ret = 'a';
        while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
            s++;
        break;
300     }
    if(eq)
        *eq = s;

305     while(s < es && strchr(whitespace, *s))
        s++;
    *ps = s;
    return ret;
}

310 int
peek(char **ps, char *es, char *toks)
{
    char *s;

315     s = *ps;
    while(s < es && strchr(whitespace, *s))
        s++;
    *ps = s;
    return *s && strchr(toks, *s);
}

```

```

}

struct cmd *parseline(char**, char*);
struct cmd *parsepipe(char**, char*);
325 struct cmd *parseexec(char**, char*);
struct cmd *nulterminate(struct cmd*);

struct cmd*
330 parsecmd(char *s)
{
    char *es;
    struct cmd *cmd;

    es = s + strlen(s);
    cmd = parseline(&s, es);
    peek(&s, es, "");
    if(s != es){
        printf(2, "leftovers: %s\n", s);
        panic("syntax");
340     }
    nulterminate(cmd);
    return cmd;
}

345 struct cmd*
parseline(char **ps, char *es)
{
    struct cmd *cmd;

350     cmd = parsepipe(ps, es);
    while(peek(ps, es, "&")){
        gettoken(ps, es, 0, 0);
        cmd = backcmd(cmd);
    }
    if(peek(ps, es, ":")){
        gettoken(ps, es, 0, 0);
        cmd = listcmd(cmd, parseline(ps, es));
    }
    return cmd;
}

360 struct cmd*
parsepipe(char **ps, char *es)
{
    struct cmd *cmd;

365     cmd = parseexec(ps, es);
    if(peek(ps, es, "|")){
        gettoken(ps, es, 0, 0);
        cmd = pipecmd(cmd, parsepipe(ps, es));
    }
    return cmd;
}

370 struct cmd*
parseredirs(struct cmd *cmd, char **ps, char *es)
{
    int tok;
    char *q, *eq;

380     while(peek(ps, es, "<>")){
        tok = gettoken(ps, es, 0, 0);
        if(gettoken(ps, es, &q, &eq) != 'a')
            panic("missing file for redirection");
    }
}

```



```

385 switch(tok){
    case '<':
        cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
        break;
    case '>':
390 cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
        break;
    case '+': // >>
        cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
        break;
395 }
    return cmd;
}

400 struct cmd*
parseblock(char **ps, char *es)
{
    struct cmd *cmd;

405 if(!peek(ps, es, "("))
    panic("parseblock");
    gettoken(ps, es, 0, 0);
    cmd = parseline(ps, es);
    if(!peek(ps, es, "("))
410 panic("syntax-missing");
    gettoken(ps, es, 0, 0);
    cmd = parseredirs(cmd, ps, es);
    return cmd;
}

415 struct cmd*
parseexec(char **ps, char *es)
{
    char *q, *eq;
420 int tok, argc;
    struct execcmd *cmd;
    struct cmd *ret;

    if(peek(ps, es, "("))
425 return parseblock(ps, es);

    ret = execcmd();
    cmd = (struct execcmd*)ret;

430 argc = 0;
    ret = parseredirs(ret, ps, es);
    while(!peek(ps, es, "|&:")){
        if((tok=gettoken(ps, es, &q, &eq)) == 0)
            break;
435 if(tok != 'a')
            panic("syntax");
        cmd->argv[argc] = q;
        cmd->eargv[argc] = eq;
        argc++;
440 if(argc >= MAXARGS)
            panic("too many args");
        ret = parseredirs(ret, ps, es);
    }
    cmd->argv[argc] = 0;
445 cmd->eargv[argc] = 0;
    return ret;
}

```

```

// NUL-terminate all the counted strings.
450 struct cmd*
nulterminate(struct cmd *cmd)
{
    int i;
    struct backcmd *bcmd;
455 struct execcmd *ecmd;
    struct listcmd *lcmd;
    struct pipecmd *pcmd;
    struct redircmd *rcmd;

460 if(cmd == 0)
    return 0;

    switch(cmd->type){
    case EXEC:
465 ecmd = (struct execcmd*)cmd;
        for(i=0; ecmd->argv[i]; i++)
            *ecmd->eargv[i] = 0;
        break;

470 case REDIR:
        rcmd = (struct redircmd*)cmd;
        nulterminate(rcmd->cmd);
        *rcmd->efile = 0;
        break;

475 case PIPE:
        pcmd = (struct pipecmd*)cmd;
        nulterminate(pcmd->left);
        nulterminate(pcmd->right);
        break;

480 case LIST:
        lcmd = (struct listcmd*)cmd;
        nulterminate(lcmd->left);
        nulterminate(lcmd->right);
        break;

    case BACK:
490 bcmd = (struct backcmd*)cmd;
        nulterminate(bcmd->cmd);
        break;
    }
    return cmd;
}

```

```

#include "types.h"
#include "x86.h"

void*
5 memset(void *dst, int c, uint n)
{
    if ((int)dst%4 == 0 && n%4 == 0){
        c &= 0xFF;
        stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
10    } else
        stosb(dst, c, n);
    return dst;
}

15 int
memcmp(const void *v1, const void *v2, uint n)
{
    const uchar *s1, *s2;

20 s1 = v1;
    s2 = v2;
    while(n-- > 0){
        if(*s1 != *s2)
            return *s1 - *s2;
25 s1++, s2++;
    }

    return 0;
}

30 void*
memmove(void *dst, const void *src, uint n)
{
    const char *s;
    char *d;

35 s = src;
    d = dst;
    if(s < d && s + n > d){
        s += n;
        d += n;
        while(n-- > 0)
            *--d = *--s;
    } else
45 while(n-- > 0)
        *d++ = *s++;

    return dst;
}

50 // memcpy exists to placate GCC. Use memmove.
void*
memcpy(void *dst, const void *src, uint n)
{
    return memmove(dst, src, n);
}

int
60 strncmp(const char *p, const char *q, uint n)
{
    while(n > 0 && *p && *q == *q)
        n--, p++, q++;
    if(n == 0)
        return 0;
}

```

```

65 return (uchar)*p - (uchar)*q;
}

char*
strncpy(char *s, const char *t, int n)
70 {
    char *os;

    os = s;
    while(n-- > 0 && (*s++ = *t++) != 0)
75 ;
    while(n-- > 0)
        *s++ = 0;
    return os;
}

80 // Like strncpy but guaranteed to NUL-terminate.
char*
safestrcpy(char *s, const char *t, int n)
{
    char *os;

    os = s;
    if(n <= 0)
        return os;
90 while(--n > 0 && (*s++ = *t++) != 0)
        ;
    *s = 0;
    return os;
}

95 int
strlen(const char *s)
{
    int n;

100 for(n = 0; s[n]; n++)
        ;
    return n;
}

105

```

Table of Contents

1	<i>types.h</i>	sheets	1 to 1	(1) pages	1- 1	5 lines
2	<i>defs.h</i>	sheets	1 to 1	(1) pages	2- 4	182 lines
3	<i>param.h</i>	sheets	2 to 2	(1) pages	5- 5	13 lines
5	4	<i>proc.h</i>	2 to 2	(1) pages	6- 7	76 lines
	5	<i>memlayout.h</i>	2 to 2	(1) pages	8- 8	23 lines
	6	<i>mmu.h</i>	3 to 3	(1) pages	9- 12	227 lines
	7	<i>x86.h</i>	4 to 4	(1) pages	13- 15	184 lines
	8	<i>asm.h</i>	4 to 4	(1) pages	16- 16	22 lines
10	9	<i>elf.h</i>	5 to 5	(1) pages	17- 17	43 lines
	10	<i>bootasm.S</i>	5 to 5	(1) pages	18- 19	89 lines
	11	<i>bootmain.c</i>	5 to 6	(2) pages	20- 21	97 lines
	12	<i>entry.S</i>	6 to 6	(1) pages	22- 22	65 lines
	13	<i>main.c</i>	6 to 6	(1) pages	23- 24	117 lines
15	14	<i>vm.c</i>	7 to 8	(2) pages	25- 30	379 lines
	15	<i>traps.h</i>	8 to 8	(1) pages	31- 31	39 lines
	16	<i>vectors.pl</i>	8 to 8	(1) pages	32- 32	48 lines
	17	<i>trapasm.S</i>	9 to 9	(1) pages	33- 33	36 lines
	18	<i>trap.c</i>	9 to 9	(1) pages	34- 35	112 lines
20	19	<i>syscall.h</i>	9 to 9	(1) pages	36- 36	23 lines
	20	<i>syscall.c</i>	10 to 10	(1) pages	37- 39	140 lines
	21	<i>sysproc.c</i>	10 to 11	(2) pages	40- 41	91 lines
	22	<i>proc.c</i>	11 to 13	(3) pages	42- 49	460 lines
	23	<i>swtch.S</i>	13 to 13	(1) pages	50- 50	29 lines
25	24	<i>kalloc.c</i>	13 to 13	(1) pages	51- 52	97 lines
	25	<i>spinlock.h</i>	14 to 14	(1) pages	53- 53	12 lines
	26	<i>spinlock.c</i>	14 to 14	(1) pages	54- 55	118 lines
	27	<i>initcode.S</i>	14 to 14	(1) pages	56- 56	32 lines
	28	<i>usys.S</i>	15 to 15	(1) pages	57- 57	32 lines
30	29	<i>init.c</i>	15 to 15	(1) pages	58- 58	38 lines
	30	<i>sh.c</i>	15 to 17	(3) pages	59- 66	495 lines
	31	<i>string.c</i>	17 to 17	(1) pages	67- 68	106 lines