types.h


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

defs.h


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

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

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

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

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

// 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);
struct inode*   idup(struct inode*);
void            iinit(void);
void            ilock(struct inode*);
void            iput(struct inode*);
void            iunlock(struct inode*);
void            iunlockput(struct inode*);
void            iupdate(struct inode*);
int             namecmp(const char*, const char*);
struct inode*   namei(char*);
struct inode*   nameiparent(char*, char*);
int             readi(struct inode*, char*, uint, uint);
void            stati(struct inode*, struct stat*);
int             writei(struct inode*, char*, uint, uint);

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

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

// kalloc.c
```

defs.h


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

// kbd.c
void            kbdintr(void);

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

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

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

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

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

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

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

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

defs.h


```
void            pushcli(void);
void            popcli(void);

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

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

// timer.c
void            timerinit(void);

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

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

// vm.c
void            seginit(void);
void            kvmalloc(void);
void            vmenable(void);
pde_t*          setupkvm(void);
char*           uva2ka(pde_t*, char*);
int             allocuvm(pde_t*, uint, uint);
int             deallocuvm(pde_t*, uint, uint);
void            freevm(pde_t*);
void            inituvm(pde_t*, char*, uint);
int             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);

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

param.h


```
#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
#define NFILE       100  // open files per system
#define NBUF         10  // size of disk block cache
#define NINODE       50  // maximum number of active i-nodes
#define NDEV         10  // maximum major device number
#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


```
// Segments in proc->gdt.
#define NSEGS     7

// Per-CPU state
struct cpu {
  uchar id;                    // Local APIC ID; index into cpus[] below
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
  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
  struct cpu *cpu;
  struct proc *proc;           // The currently-running process.
};

extern struct cpu cpus[NCPU];
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
// %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.
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.
// 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
// describe; the stack pointer is the address of the context.
// The layout of the context matches the layout of the stack in swtch.S
// at the "Switch stacks" comment. Switch doesn't save eip explicitly,
// but it is on the stack and allocproc() manipulates it.
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};

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

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

proc.h


```
  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)
};

// Process memory is laid out contiguously, low addresses first:
//   text
//   original data and bss
//   fixed-size stack
//   expandable heap
```

memlayout.h


```
// Memory layout

#define EXTMEM  0x100000            // Start of extended memory
#define PHYSTOP 0xE000000           // Top physical memory
#define DEVSPACE 0xFE000000         // 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
#define KERNLINK (KERNBASE+EXTMEM)  // Address where kernel is linked

#ifndef __ASSEMBLER__

static inline uint v2p(void *a) { return ((uint) (a))  - KERNBASE; }
static inline void *p2v(uint a) { return (void *) ((a) + KERNBASE); }

#endif

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

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

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

// Eflags register
#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
#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
#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
#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
#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
#define CR0_EM          0x00000004      // Emulation
#define CR0_TS          0x00000008      // Task Switched
#define CR0_ET          0x00000010      // Extension Type
#define CR0_NE          0x00000020      // Numeric Errror
#define CR0_WP          0x00010000      // Write Protect
#define CR0_AM          0x00040000      // Alignment Mask
#define CR0_NW          0x20000000      // Not Writethrough
#define CR0_CD          0x40000000      // Cache Disable
#define CR0_PG          0x80000000      // Paging

#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
#define SEG_UCODE 4  // user code
#define SEG_UDATA 5  // user data+stack
#define SEG_TSS   6  // this process's task state

//PAGEBREAK!
#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
  uint 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
  uint 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
```



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

// Normal segment
#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,              \
  ((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

// 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)
#define STA_W       0x2     // Writeable (non-executable segments)
#define STA_R       0x2     // Readable (executable segments)
#define STA_A       0x1     // Accessed

// System segment type bits
#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
#define STS_TG      0x5     // Task Gate / Coum Transmitions
#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
#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:
//
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table   | Offset within Page  |
// |      Index     |      Index     |                     |
// +----------------+----------------+---------------------+
//  \--- PDX(va) --/ \--- PTX(va) --/

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

// page table index
#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)))

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

#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
```



```
#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   // Writeable
#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
#define PTE_PS          0x080   // Page Size
#define PTE_MBZ         0x180   // Bits must be zero

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

#ifndef __ASSEMBLER__
typedef uint pte_t;

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

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



```
struct gatedesc {
  uint 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})
  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
};

// 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
// - 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)                \
{                                                         \
  (gate).off_15_0 = (uint)(off) & 0xffff;                \
  (gate).cs = (sel);                                      \
  (gate).args = 0;                                        \
  (gate).rsv1 = 0;                                        \
  (gate).type = (istrap) ? STS_TG32 : STS_IG32;           \
  (gate).s = 0;                                           \
  (gate).dpl = (d);                                       \
  (gate).p = 1;                                           \
  (gate).off_31_16 = (uint)(off) >> 16;                   \
}

#endif
```

x86.h Page 1/3

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

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

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

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

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

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

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

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

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

struct segdesc;

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

x86.h Page 2/3

```
  volatile ushort pd[3];

  pd[0] = size-1;
  pd[1] = (uint)p;
  pd[2] = (uint)p >> 16;

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

struct gatedesc;

static inline void
lidt(struct gatedesc *p, int size)
{
  volatile ushort pd[3];

  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)
{
  asm volatile("ltr %0" : : "r" (sel));
}

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

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

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

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

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

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

x86.h Page 3/3

```
               "cc");
  return result;
}

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

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

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

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

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

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

asm.h Page 1/1

```
//
// assembler macros to create x86 segments
//

#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.
#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)

#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)
#define STA_R     0x2       // Readable (executable segments)
#define STA_A     0x1       // Accessed
```

elf.h


```
// Format of an ELF executable file

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

// 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
struct proghdr {
  uint type;
  uint off;
  uint vaddr;
  uint paddr;
  uint filesz;
  uint memsz;
  uint flags;
  uint align;
};

// Values for Proghdr type
#define ELF_PROG_LOAD           1

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

bootasm.S


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

# 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.

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

  # 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

  # 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

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

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

  # Switch from real to protected mode.  Use a bootstrap GDT that makes
  # 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
  movl    %eax, %cr0

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

.code32  # Tell assembler to generate 32-bit code now.
start32:
  # Set up the protected-mode data segment registers
  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
  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.
```

bootasm.S


```
  movl    $start, %esp
  call    bootmain

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

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

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

bootmain.c


```
// 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.
// 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"
#include "x86.h"
#include "memlayout.h"

#define SECTSIZE  512

void readseg(uchar*, uint, uint);

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

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

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

  // 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)
      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();
}

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

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

bootmain.c


```
  outb(0x1F3, offset);
  outb(0x1F4, offset >> 8);
  outb(0x1F5, offset >> 16);
  outb(0x1F6, (offset >> 24) | 0xE0);
  outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

  // Read data.
  waitdisk();
  insl(0x1F0, dst, SECTSIZE/4);
}

// Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
// Might copy more than asked.
void
readseg(uchar* pa, uint count, uint offset)
{
  uchar* epa;

  epa = pa + count;

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

  // Translate from bytes to sectors; kernel starts at sector 1.
  offset = (offset / SECTSIZE) + 1;

  // 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++)
    readsect(pa, offset);
}
```

entry.S


```
# 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
# Linux file system by copying kernel or kernelmemfs to /boot
# and then adding this menu entry:
#
# menuentry "xv6" {
#   insmod ext2
#   set root='(hd0,msdos1)'
#   set kernel='/boot/kernel'
#   echo "Loading ${kernel}..."
#   multiboot ${kernel} ${kernel}
#   boot
# }

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

# Multiboot header.  Data to direct multiboot loader.
.p2align 2
.text
.globl multiboot_header
multiboot_header:
  #define magic 0x1badb002
  #define flags 0
  .long magic
  .long flags
  .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
# the physical address of 'entry'.
.globl _start
_start = V2P_WO(entry)

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

  # Set up the stack pointer.
  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.
  mov $main, %eax
  jmp *%eax

.comm stack, KSTACKSIZE
```

main.c


```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#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

// 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)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // collect info about this machine
  lapicinit();
  seginit();       // set up segments
  cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
  picinit();       // interrupt controller
  ioapicinit();    // another interrupt controller
  consoleinit();   // I/O devices & their interrupts
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  iinit();         // inode cache
  ideinit();       // disk
  if(!ismp)
    timerinit();   // uniprocessor timer
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  // Finish setting up this processor in mpmain.
  mpmain();
}

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

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

main.c


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

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

  // 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);

  for(c = cpus; c < cpus+ncpu; c++){
    if(c == cpus+cpunum())  // We've started already.
      continue;

    // 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);

    lapicstartap(c->id, v2p(code));

    // wait for cpu to finish mpmain()
    while(c->started == 0)
      ;
  }
}

// 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.
// 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,
  // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
  [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};

//PAGEBREAK!
// 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
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;
}

// 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;
  uint phys_end;
  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
switchuvm(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("switchuvm: 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
inituvm(pde_t *pgdir, char *init, uint sz)
{
  char *mem;

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

vm.c


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

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

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

  // Clear PTE_U on a page. Used to create an inaccessible
  // page beneath the user stack.
  void
  clearpteu(pde_t *pgdir, char *uva)
  {
    pte_t *pte;

    pte = walkpgdir(pgdir, uva, 0);
    if(pte == 0)
      panic("clearpteu");
    *pte &= ~PTE_U;
  }

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

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

vm.c


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

  bad:
    freevm(d);
    return 0;
  }

  //PAGEBREAK!
  // Map user virtual address to kernel address.
  char*
  uva2ka(pde_t *pgdir, char *uva)
  {
    pte_t *pte;

    pte = walkpgdir(pgdir, uva, 0);
    if((*pte & PTE_P) == 0)
      return 0;
    if((*pte & PTE_U) == 0)
      return 0;
    return (char*)p2v(PTE_ADDR(*pte));
  }

  // Copy len bytes from p to user address va in page table pgdir.
  // 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)
  {
    char *buf, *pa0;
    uint n, va0;

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

traps.h


```
  // x86 trap and interrupt constants.

  // Processor-defined:
  #define T_DIVIDE         0      // divide error
  #define T_DEBUG          1      // debug exception
  #define T_NMI            2      // non-maskable interrupt
  #define T_BRKPT          3      // breakpoint
  #define T_OFLOW          4      // overflow
  #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)
  #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
  // #define T_RES        15      // reserved
  #define T_FPERR         16      // floating point error
  #define T_ALIGN         17      // aligment check
  #define T_MCHK          18      // machine check
  #define T_SIMDERR       19      // SIMD floating point error

  // 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

  #define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ

  #define IRQ_TIMER        0
  #define IRQ_KBD          1
  #define IRQ_COM1         4
  #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
  # 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";
  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) || $i == 17)){
          print "  pushl \$0\n";
      }
      print "  pushl \$$i\n";
      print "  jmp alltraps\n";
  }

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

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

trapasm.S


```
#include "mmu.h"

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

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

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

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

trap.c


```
#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.
```

trap.c


```
    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();
}
```

syscall.h


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

syscall.c Page 1/3

```
#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;
```

syscall.c Page 2/3

```
}

// 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)
{
```

syscall.c Page 3/3

```
  int num;

  num = proc->tf->eax;
  if(num > 0 && 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;
  }
}
```

sysproc.c Page 1/2

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

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

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

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

int
sys_kill(void)
{
  int pid;

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

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

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

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

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

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

sysproc.c — Page 2/2

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

// return how many clock tick interrupts have occurred
// since start.
int
sys_uptime(void)
{
  uint xticks;

  acquire(&tickslock);
  xticks = ticks;
  release(&tickslock);
  return xticks;
}
```

proc.c — Page 1/8

```
#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.
```

proc.c — Page 2/8

```
  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()) == 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
```

proc.c — Page 3/8

```
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 = "???";
```

proc.c


```
      cprintf("%d %s %s", p->pid, state, p->name);
450   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");
    }
  }
```

swtch.S


```
  # 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
    pushl %ebp
15  pushl %ebx
    pushl %esi
    pushl %edi

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

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

kalloc.c


```
  // 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

15struct 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.
30void
  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;
  }

45void
  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
60kfree(char *v)
  {
    struct run *r;

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

kalloc.c


```
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.
80// 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}
```

spinlock.h


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

  // 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.
};
```

spinlock.c


```
// 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();
}
```

spinlock.c


```

// 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();
}
```

initcode.S


```
# Initial process execs /init.

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


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

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

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

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

usys.S


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

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

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
```

init.c


```
// init: The initial user-level program

#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");
  }
}
```

sh.c


```
// 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;
```

sh.c


```
  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;
}

int
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;
    }
  }

  // Read and run input commands.
  while(getcmd(buf, sizeof(buf)) >= 0){
    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)
        printf(2, "cannot cd %s\n", buf+3);
      continue;
    }
    if(fork1() == 0)
      runcmd(parsecmd(buf));
    wait();
  }
  exit();
}

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

int
fork1(void)
{
  int pid;

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



```
//PAGEBREAK!
// Constructors

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

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

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

  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;
  return (struct cmd*)cmd;
}

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

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

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

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

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



```
  cmd->cmd = subcmd;
  return (struct cmd*)cmd;
}
//PAGEBREAK!
// Parsing

char whitespace[] = " \t\r\n\v";
char symbols[] = "<|>&;()";

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

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

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

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

  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*);
struct cmd *parseexec(char**, char*);
struct cmd *nulterminate(struct cmd*);

struct cmd*
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");
  }
  nulterminate(cmd);
  return cmd;
}

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

  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;
}

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

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

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

  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);
480       break;

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

        case BACK:
          bcmd = (struct backcmd*)cmd;
490       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;
35      char *d;

        s = src;
        d = dst;
        if(s < d && s + n > d){
40        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)
      {
55      return memmove(dst, src, n);
      }

      int
      strncmp(const char *p, const char *q, uint n)
60    {
        while(n > 0 && *p && *p == *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)
      {
85      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 Content

Table of Contents