xv6は、Dennis RitchieとKen ThompsonのUnix Version 6 (v6) の再実装である。
xv6は、おおよそv6の構造とスタイルに従っているが、ANSI Cを使用し、最新の
x86ベースのマルチプロセッサ向けに実装されている。

謝辞

xv6は、John LionsのUNIX 6th Commentary (Peer to Peer Communications;
ISBN：1-57398-013-7; 1st edition (June 14、2000)) に触発されたものである。
http://pdos.csail.mit.edu/6.828/2016/xv6.htmlも参照されたい。これには、
v6に関するオンラインリソースをまとめられている。

xv6は次のソースからコードを拝借している:
     JOS (asm.h, elf.h, mmu.h, bootasm.S, ide.c, console.c, など)
     Plan 9 (entryother.S, mp.h, mp.c, lapic.c)
     FreeBSD (ioapic.c)
     NetBSD (console.c)

次の方々に感謝いたします: Russ Cox (コンテキストスイッチ、ロック) 、
Cliff Frey (MP)、Xiao Yu (MP)、Nickolai Zeldovich、Austin Clements.

バグの報告とパッチをいただいた次の方々にも感謝いたします: Silas
Boyd-Wickizer, Anton Burtsev, Cody Cutler, Mike CAT, Tej Chajed, Nelson
Elhage, Saar Ettinger, Alice Ferrazzi, Nathaniel Filardo, Peter Froehlich,
Yakir Goaron, Shivam Handa, Bryan Henry, Jim Huang, Alexander Kapshuk,
Anders Kaseorg, kehao95, Wolfgang Keller, Eddie Kohler, Austin Liew,
Imbar Marinescu, Yandong Mao, Hitoshi Mitake, Carmi Merimovich, Joel Nider,
Greg Price, Ayan Shafqat, Eldar Sehayek, Yongming Shen, Cam Tenny,
Rafael Ubal, Warren Toomey, Stephen Tu, Pablo Ventura, Xi Wang, Keiichi
Watanabe, Nicolas Wolovick, Grant Wu, Jindong Zhang, Icenowy Zheng,
Zou Chang Wei.

xv6を構成するファイルのコードの著作権表ゔは次のとおりである。

エラー報告

エラーや提案をFrans KaashoekとRobert Morris (kaashoek、rtm @ mit.edu) に
送ってください。xv6の主たる目的は、MITの講義科目6.828のための教育用
オペレーティングシステムである。そのため、我々は新しい機能よりも単純化と
明確化に関心がある。

XV6のビルドと実行

xv6をx86 ELFマシン (LinuxやFreeBSDなど) でビルドするには、"make"を実行する。
非x86マシンあるいは (たとえx86でもOS Xなどの) ELF以外のマシンでは、x86 ELF
バイナリを生成できるクロスコンパイラgccスイートをインストールする必要がある。
http://pdos.csail.mit.edu/6.828/2016/tools.htmlを参照されたい。
インストール後、 "make TOOLPREFIX=i386-jos-elf-"を実行する。 そして、
QEMU PCシミュレータをインストールし、"make qemu"を実行する。

表中のファイル名の左側の数字はシート番号である。ソースコードは、ページ
当たり2カラム、1カラムは50行で印刷されており、1シート (1ページ) あたり
100行表ゔされている。
行番号とシート番号には、行番号の先頭2桁がシート番号という便利な関係がある。

ソースリストの前には、xv6で定義されているすべての定数、構造体、グローバル
変数、関数の相互参照表がある。参照項目には、名前とその名前が定義されている
行番号 (複数の行番号がある場合もある) が同じ行にゔされている。参照項目の
2行目以下には、その名前が使用されている行番号がゔされている。
たとえば、次の参照項目は

     swtch 2658
          0374 2428 2466 2657 2658

swtchは2658行で定義されており、シート03と24, 26の5つの行で使用されている
ことをゔしている。

```
0100 typedef unsigned int   uint;
0101 typedef unsigned short ushort;
0102 typedef unsigned char  uchar;
0103 typedef uint pde_t;
0104
0105
0106
0107
0108
0109
0110
0111
0112
0113
0114
0115
0116
0117
0118
0119
0120
0121
0122
0123
0124
0125
0126
0127
0128
0129
0130
0131
0132
0133
0134
0135
0136
0137
0138
0139
0140
0141
0142
0143
0144
0145
0146
0147
0148
0149
```



```
0150 #define NPROC        64  // プロセスの最大数
0151 #define KSTACKSIZE 4096  // 各プロセスのカーネルスタックサイズ
0152 #define NCPU          8  // CPUの最大数
0153 #define NOFILE       16  // プロセス当りのオープンファイル
0154 #define NFILE       100  // システム当りのオープンファイル
0155 #define NINODE       50  // アクティブinodeの最大数
0156 #define NDEV         10  // メジャーデバイス番号の最大値
0157 #define ROOTDEV       1  // ルートディスクファイルシステムのデバイス番号
0158 #define MAXARG       32  // execの引数の最大数
0159 #define MAXOPBLOCKS  10  // FS操作関数が書き込み可能な最大ブロック数
0160 #define LOGSIZE      (MAXOPBLOCKS*3)  // オンディスクログの最大データブロック
0161 #define NBUF         (MAXOPBLOCKS*3)  // ディスクブロックキャッシュのサイズ
0162 #define FSSIZE       1000  // ファイルシステムのサイズ（単位はブロック）
0163
0164
0165
0166
0167
0168
0169
0170
0171
0172
0173
0174
0175
0176
0177
0178
0179
0180
0181
0182
0183
0184
0185
0186
0187
0188
0189
0190
0191
0192
0193
0194
0195
0196
0197
0198
0199
```

```
0200 // メモリレイアウト
0201
0202 #define EXTMEM  0x100000            // 拡張メモリの開始アドレス
0203 #define PHYSTOP 0xE000000           // 物理メモリの最上位アドレス
0204 #define DEVSPACE 0xFE000000         // その他のデバイスは高位アドレスにある
0205
0206 // アドレス空間レイアウトの主要なアドレス（レイアウトはvm.cのkmapを参照）
0207 #define KERNBASE 0x80000000         // カーネル仮想アドレスの開始アドレス
0208 #define KERNLINK (KERNBASE+EXTMEM)  // カーネルのリンク先アドレス
0209
0210 #define V2P(a) (((uint) (a)) - KERNBASE)
0211 #define P2V(a) (((void *) (a)) + KERNBASE)
0212
0213 #define V2P_WO(x) ((x) - KERNBASE)    // V2Pと同じだが、キャストはしない
0214 #define P2V_WO(x) ((x) + KERNBASE)    // P2Vと同じだが、キャストはしない
0215
0216
0217
0218
0219
0220
0221
0222
0223
0224
0225
0226
0227
0228
0229
0230
0231
0232
0233
0234
0235
0236
0237
0238
0239
0240
0241
0242
0243
0244
0245
0246
0247
0248
0249
```



```
0250 struct buf;
0251 struct context;
0252 struct file;
0253 struct inode;
0254 struct pipe;
0255 struct proc;
0256 struct rtcdate;
0257 struct spinlock;
0258 struct sleeplock;
0259 struct stat;
0260 struct superblock;
0261
0262 // bio.c
0263 void            binit(void);
0264 struct buf*     bread(uint, uint);
0265 void            brelse(struct buf*);
0266 void            bwrite(struct buf*);
0267
0268 // console.c
0269 void            consoleinit(void);
0270 void            cprintf(char*, ...);
0271 void            consoleintr(int(*)(void));
0272 void            panic(char*) __attribute__((noreturn));
0273
0274 // exec.c
0275 int             exec(char*, char**);
0276
0277 // file.c
0278 struct file*    filealloc(void);
0279 void            fileclose(struct file*);
0280 struct file*    filedup(struct file*);
0281 void            fileinit(void);
0282 int             fileread(struct file*, char*, int n);
0283 int             filestat(struct file*, struct stat*);
0284 int             filewrite(struct file*, char*, int n);
0285
0286 // fs.c
0287 void            readsb(int dev, struct superblock *sb);
0288 int             dirlink(struct inode*, char*, uint);
0289 struct inode*   dirlookup(struct inode*, char*, uint*);
0290 struct inode*   ialloc(uint, short);
0291 struct inode*   idup(struct inode*);
0292 void            iinit(int dev);
0293 void            ilock(struct inode*);
0294 void            iput(struct inode*);
0295 void            iunlock(struct inode*);
0296 void            iunlockput(struct inode*);
0297 void            iupdate(struct inode*);
0298 int             namecmp(const char*, const char*);
0299 struct inode*   namei(char*);
```

```
0300 struct inode*   nameiparent(char*, char*);
0301 int             readi(struct inode*, char*, uint, uint);
0302 void            stati(struct inode*, struct stat*);
0303 int             writei(struct inode*, char*, uint, uint);
0304
0305 // ide.c
0306 void            ideinit(void);
0307 void            ideintr(void);
0308 void            iderw(struct buf*);
0309
0310 // ioapic.c
0311 void            ioapicenable(int irq, int cpu);
0312 extern uchar    ioapicid;
0313 void            ioapicinit(void);
0314
0315 // kalloc.c
0316 char*           kalloc(void);
0317 void            kfree(char*);
0318 void            kinit1(void*, void*);
0319 void            kinit2(void*, void*);
0320
0321 // kbd.c
0322 void            kbdintr(void);
0323
0324 // lapic.c
0325 void            cmostime(struct rtcdate *r);
0326 int             lapicid(void);
0327 extern volatile uint*    lapic;
0328 void            lapiceoi(void);
0329 void            lapicinit(void);
0330 void            lapicstartap(uchar, uint);
0331 void            microdelay(int);
0332
0333 // log.c
0334 void            initlog(int dev);
0335 void            log_write(struct buf*);
0336 void            begin_op();
0337 void            end_op();
0338
0339 // mp.c
0340 extern int      ismp;
0341 void            mpinit(void);
0342
0343 // picirq.c
0344 void            picenable(int);
0345 void            picinit(void);
0346
0347
0348
0349
```



```
0350 // pipe.c
0351 int             pipealloc(struct file**, struct file**);
0352 void            pipeclose(struct pipe*, int);
0353 int             piperead(struct pipe*, char*, int);
0354 int             pipewrite(struct pipe*, char*, int);
0355
0356
0357 // proc.c
0358 int             cpuid(void);
0359 void            exit(void);
0360 int             fork(void);
0361 int             growproc(int);
0362 int             kill(int);
0363 struct cpu*     mycpu(void);
0364 struct proc*    myproc();
0365 void            pinit(void);
0366 void            procdump(void);
0367 void            scheduler(void) __attribute__((noreturn));
0368 void            sched(void);
0369 void            setproc(struct proc*);
0370 void            sleep(void*, struct spinlock*);
0371 void            userinit(void);
0372 int             wait(void);
0373 void            wakeup(void*);
0374 void            yield(void);
0375
0376 // swtch.S
0377 void            swtch(struct context**, struct context*);
0378
0379 // spinlock.c
0380 void            acquire(struct spinlock*);
0381 void            getcallerpcs(void*, uint*);
0382 int             holding(struct spinlock*);
0383 void            initlock(struct spinlock*, char*);
0384 void            release(struct spinlock*);
0385 void            pushcli(void);
0386 void            popcli(void);
0387
0388 // sleeplock.c
0389 void            acquiresleep(struct sleeplock*);
0390 void            releasesleep(struct sleeplock*);
0391 int             holdingsleep(struct sleeplock*);
0392 void            initsleeplock(struct sleeplock*, char*);
0393
0394 // string.c
0395 int             memcmp(const void*, const void*, uint);
0396 void*           memmove(void*, const void*, uint);
0397 void*           memset(void*, int, uint);
0398 char*           safestrcpy(char*, const char*, int);
0399 int             strlen(const char*);
```

```
0400 int             strncmp(const char*, const char*, uint);
0401 char*           strncpy(char*, const char*, int);
0402
0403 // syscall.c
0404 int             argint(int, int*);
0405 int             argptr(int, char**, int);
0406 int             argstr(int, char**);
0407 int             fetchint(uint, int*);
0408 int             fetchstr(uint, char**);
0409 void            syscall(void);
0410
0411 // timer.c
0412 void            timerinit(void);
0413
0414 // trap.c
0415 void            idtinit(void);
0416 extern uint     ticks;
0417 void            tvinit(void);
0418 extern struct spinlock tickslock;
0419
0420 // uart.c
0421 void            uartinit(void);
0422 void            uartintr(void);
0423 void            uartputc(int);
0424
0425 // vm.c
0426 void            seginit(void);
0427 void            kvmalloc(void);
0428 pde_t*          setupkvm(void);
0429 char*           uva2ka(pde_t*, char*);
0430 int             allocuvm(pde_t*, uint, uint);
0431 int             deallocuvm(pde_t*, uint, uint);
0432 void            freevm(pde_t*);
0433 void            inituvm(pde_t*, char*, uint);
0434 int             loaduvm(pde_t*, char*, struct inode*, uint, uint);
0435 pde_t*          copyuvm(pde_t*, uint);
0436 void            switchuvm(struct proc*);
0437 void            switchkvm(void);
0438 int             copyout(pde_t*, uint, void*, uint);
0439 void            clearpteu(pde_t *pgdir, char *uva);
0440
0441 // 固定サイズの配列の要素数
0442 #define NELEM(x) (sizeof(x)/sizeof((x)[0]))
0443
0444
0445
0446
0447
0448
0449
```



```
0450 // Cコードでx86特殊命令を使用するためのルーチン。
0451
0452 static inline uchar
0453 inb(ushort port)
0454 {
0455   uchar data;
0456
0457   asm volatile("in %1,%0" : "=a" (data) : "d" (port));
0458   return data;
0459 }
0460
0461 static inline void
0462 insl(int port, void *addr, int cnt)
0463 {
0464   asm volatile("cld; rep insl" :
0465                "=D" (addr), "=c" (cnt) :
0466                "d" (port), "0" (addr), "1" (cnt) :
0467                "memory", "cc");
0468 }
0469
0470 static inline void
0471 outb(ushort port, uchar data)
0472 {
0473   asm volatile("out %0,%1" : : "a" (data), "d" (port));
0474 }
0475
0476 static inline void
0477 outw(ushort port, ushort data)
0478 {
0479   asm volatile("out %0,%1" : : "a" (data), "d" (port));
0480 }
0481
0482 static inline void
0483 outsl(int port, const void *addr, int cnt)
0484 {
0485   asm volatile("cld; rep outsl" :
0486                "=S" (addr), "=c" (cnt) :
0487                "d" (port), "0" (addr), "1" (cnt) :
0488                "cc");
0489 }
0490
0491 static inline void
0492 stosb(void *addr, int data, int cnt)
0493 {
0494   asm volatile("cld; rep stosb" :
0495                "=D" (addr), "=c" (cnt) :
0496                "0" (addr), "1" (cnt), "a" (data) :
0497                "memory", "cc");
0498 }
0499
```

```
0500 static inline void
0501 stosl(void *addr, int data, int cnt)
0502 {
0503   asm volatile("cld; rep stosl" :
0504                "=D" (addr), "=c" (cnt) :
0505                "0" (addr), "1" (cnt), "a" (data) :
0506                "memory", "cc");
0507 }
0508
0509 struct segdesc;
0510
0511 static inline void
0512 lgdt(struct segdesc *p, int size)
0513 {
0514   volatile ushort pd[3];
0515
0516   pd[0] = size-1;
0517   pd[1] = (uint)p;
0518   pd[2] = (uint)p >> 16;
0519
0520   asm volatile("lgdt (%0)" : : "r" (pd));
0521 }
0522
0523 struct gatedesc;
0524
0525 static inline void
0526 lidt(struct gatedesc *p, int size)
0527 {
0528   volatile ushort pd[3];
0529
0530   pd[0] = size-1;
0531   pd[1] = (uint)p;
0532   pd[2] = (uint)p >> 16;
0533
0534   asm volatile("lidt (%0)" : : "r" (pd));
0535 }
0536
0537 static inline void
0538 ltr(ushort sel)
0539 {
0540   asm volatile("ltr %0" : : "r" (sel));
0541 }
0542
0543 static inline uint
0544 readeflags(void)
0545 {
0546   uint eflags;
0547   asm volatile("pushfl; popl %0" : "=r" (eflags));
0548   return eflags;
0549 }
```



```
0550 static inline void
0551 loadgs(ushort v)
0552 {
0553   asm volatile("movw %0, %%gs" : : "r" (v));
0554 }
0555
0556 static inline void
0557 cli(void)
0558 {
0559   asm volatile("cli");
0560 }
0561
0562 static inline void
0563 sti(void)
0564 {
0565   asm volatile("sti");
0566 }
0567
0568 static inline uint
0569 xchg(volatile uint *addr, uint newval)
0570 {
0571   uint result;
0572
0573   // "+m"の+はread-modify-writeオペランドをぉす
0574   asm volatile("lock; xchgl %0, %1" :
0575                "+m" (*addr), "=a" (result) :
0576                "1" (newval) :
0577                "cc");
0578   return result;
0579 }
0580
0581 static inline uint
0582 rcr2(void)
0583 {
0584   uint val;
0585   asm volatile("movl %%cr2,%0" : "=r" (val));
0586   return val;
0587 }
0588
0589 static inline void
0590 lcr3(uint val)
0591 {
0592   asm volatile("movl %0,%%cr3" : : "r" (val));
0593 }
0594
0595
0596
0597
0598
0599
```

```
0600 // ハードウェアとtrapasm.Sによりスタック上に構築され、
0601 // trap()に渡されるトラップフレームのレイアウト
0602 struct trapframe {
0603   // pushaによりプッシュされるレジスタ群
0604   uint edi;
0605   uint esi;
0606   uint ebp;
0607   uint oesp;      // 役立たずで無視される
0608   uint ebx;
0609   uint edx;
0610   uint ecx;
0611   uint eax;
0612
0613   // トラップフレームのその他の部分
0614   ushort gs;
0615   ushort padding1;
0616   ushort fs;
0617   ushort padding2;
0618   ushort es;
0619   ushort padding3;
0620   ushort ds;
0621   ushort padding4;
0622   uint trapno;
0623
0624   // これより下はx86ハードウェアにより定義されている
0625   uint err;
0626   uint eip;
0627   ushort cs;
0628   ushort padding5;
0629   uint eflags;
0630
0631   // これより下はユーザからカーネルなど、空間をまたぐ時のみ
0632   uint esp;
0633   ushort ss;
0634   ushort padding6;
0635 };
0636
0637
0638
0639
0640
0641
0642
0643
0644
0645
0646
0647
0648
0649
```



```
0650 //
0651 // x86セグメントを作成するためのアセンブラマクロ
0652 // ( 訳注: セグメントディスクリプタ(32 bit)を作成する )
0653
0654 #define SEG_NULLASM                                             \
0655         .word 0, 0;                                             \
0656         .byte 0, 0, 0, 0
0657
0658 // 0xC0はlimitが4096バイト単位であり、 ( 実行セグメントは )
0659 // 32ビットモードであることを意味する(訳注: G & D bitをオン)
0660 #define SEG_ASM(type,base,lim)                                  \
0661         .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);      \
0662         .byte (((base) >> 16) & 0xff), (0x90 ¦ (type)),         \
0663                 (0xC0 ¦ (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
0664
0665 #define STA_X     0x8       // 実行セグメント
0666 #define STA_E     0x4       // 拡大縮小 ( 非実行セグメント )
0667 #define STA_C     0x4       // コンフォーミングコードセグメント ( 実行のみ )
0668 #define STA_W     0x2       // 書き込み可能 ( 非実行セグメント )
0669 #define STA_R     0x2       // 読み取り可能 ( 実行セグメント )
0670 #define STA_A     0x1       // アクセス
0671
0672
0673
0674
0675
0676
0677
0678
0679
0680
0681
0682
0683
0684
0685
0686
0687
0688
0689
0690
0691
0692
0693
0694
0695
0696
0697
0698
0699
```

```
0700 // このファイルはx86メモリ管理ユニット(MMU)
0701 // のための定義を含んでいる。
0702
0703 // Eflagsレジスタ
0704 #define FL_CF           0x00000001      //  1: キャリーフラグ
0705 #define FL_PF           0x00000004      //  2: パリティフラグ
0706 #define FL_AF           0x00000010      //  4: 補助キャリーフラグ
0707 #define FL_ZF           0x00000040      //  6: ゼロフラグ
0708 #define FL_SF           0x00000080      //  7: サインフラグ
0709 #define FL_TF           0x00000100      //  8: トラップフラグ
0710 #define FL_IF           0x00000200      //  9: 割り込みを有効化
0711 #define FL_DF           0x00000400      // 10: ディレクションフラグ
0712 #define FL_OF           0x00000800      // 11: オーバーフローフラグ
0713 #define FL_IOPL_MASK    0x00003000      // 12-13: I/O特権レベル・ビットマスク
0714 #define FL_IOPL_0       0x00000000      //   IOPL == 0
0715 #define FL_IOPL_1       0x00001000      //   IOPL == 1
0716 #define FL_IOPL_2       0x00002000      //   IOPL == 2
0717 #define FL_IOPL_3       0x00003000      //   IOPL == 3
0718 #define FL_NT           0x00004000      // 14: ネストタスク
0719 #define FL_RF           0x00010000      // 16: レジュームフラグ
0720 #define FL_VM           0x00020000      // 17: 仮想8086モード
0721 #define FL_AC           0x00040000      // 18: アライメントチェック
0722 #define FL_VIF          0x00080000      // 19: 仮想割り込みフラグ
0723 #define FL_VIP          0x00100000      // 20: 仮想割り込みペンディング
0724 #define FL_ID           0x00200000      // 21: IDフラグ
0725
0726 // コントロールレジスタフラグ
0727 #define CR0_PE          0x00000001      //  0: プロテクトモードを有効化
0728 #define CR0_MP          0x00000002      //  1: モニタコプロセッサ
0729 #define CR0_EM          0x00000004      //  2: エミュレーション
0730 #define CR0_TS          0x00000008      //  3: タスクスイッチ
0731 #define CR0_ET          0x00000010      //  4: 拡張タイプ
0732 #define CR0_NE          0x00000020      //  5: 数値演算エラー
0733 #define CR0_WP          0x00010000      // 16: 書き込み保護
0734 #define CR0_AM          0x00040000      // 18: アライメントマスク
0735 #define CR0_NW          0x20000000      // 29: ノットライトスルー
0736 #define CR0_CD          0x40000000      // 30: キャッシュ無効化
0737 #define CR0_PG          0x80000000      // 31: ページング
0738
0739 #define CR4_PSE         0x00000010      // 4: ページサイズ拡張
0740
0741 // セグメントセレクタ
0742 #define SEG_KCODE 1  // カーネルコード
0743 #define SEG_KDATA 2  // カーネルデータとスタック
0744 #define SEG_UCODE 3  // ユーザコード
0745 #define SEG_UDATA 4  // ユーザデータとスタック
0746 #define SEG_TSS   5  // このプロセスのタスクステート
0747
0748 // cpu->gdt[NSEGS] は上記のセグメントを保持する
0749 #define NSEGS     6
```



```
0750 #ifndef __ASSEMBLER__
0751 // セグメントディスクリプタ
0752 struct segdesc {
0753   uint lim_15_0 : 16;  // セグメントリミット値の低位ビット
0754   uint base_15_0 : 16; // セグメントベースアドレスの低位ビット
0755   uint base_23_16 : 8; // セグメントベースアドレスの中位ビット
0756   uint type : 4;       // セグメントタイプ ( STS_ 定数を参照 )
0757   uint s : 1;          // 0 = システム, 1 = アプリケーション
0758   uint dpl : 2;        // ディスクリプタの特権レベル
0759   uint p : 1;          // 存在する
0760   uint lim_19_16 : 4;  // セグメントリミット値の高位ビット
0761   uint avl : 1;        // 未使用 ( ソフトウェアで使用可 )
0762   uint rsv1 : 1;       // 予約済
0763   uint db : 1;         // 0 = 16 ビットセグメント, 1 = 32 ビットセグメント
0764   uint g : 1;          // 単位: セットされるとリミット値が4K倍される
0765   uint base_31_24 : 8; // セグメントベースアドレスの高位ビット
0766 };
0767
0768 // 通常のセグメント
0769 #define SEG(type, base, lim, dpl) (struct segdesc)    \
0770 { ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff,      \
0771   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
0772   (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
0773 #define SEG16(type, base, lim, dpl) (struct segdesc)  \
0774 { (lim) & 0xffff, (uint)(base) & 0xffff,              \
0775   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
0776   (uint)(lim) >> 16, 0, 0, 1, 0, (uint)(base) >> 24 }
0777 #endif
0778
0779 #define DPL_USER    0x3     // ユーザDPL
0780
0781 // アプリケーションセグメントのタイプビット
0782 #define STA_X       0x8     // 実行セグメント
0783 #define STA_E       0x4     // 拡大縮小 ( 非実行セグメント )
0784 #define STA_C       0x4     // コンフォーミングコードセグメント ( 実行のみ )
0785 #define STA_W       0x2     // 書き込み可能 ( 非実行セグメント )
0786 #define STA_R       0x2     // 読み取り可能 ( 実行セグメント )
0787 #define STA_A       0x1     // アクセス
0788
0789 // システムセグメントのタイプビット
0790 #define STS_T16A    0x1     // 16ビットTSSを利用可能
0791 #define STS_LDT     0x2     // ローカルディスクリプタテーブル
0792 #define STS_T16B    0x3     // ビジーな16ビットTSS
0793 #define STS_CG16    0x4     // 16ビットコールゲート
0794 #define STS_TG      0x5     // タスクゲート / Coum Transmitions
0795 #define STS_IG16    0x6     // 16ビット割り込みゲート
0796 #define STS_TG16    0x7     // 16ビットトラップゲート
0797 #define STS_T32A    0x9     // 32ビットTSSを利用可能
0798 #define STS_T32B    0xB     // ビジーな32ビットTSS
0799 #define STS_CG32    0xC     // 32ビットコールゲート
```

```
0800 #define STS_IG32    0xE     // 32ビット割り込みゲート
0801 #define STS_TG32    0xF     // 32ビットトラップゲート
0802
0803 // 仮想アドレス 'la' は次の3要素からなる構造をもつ:
0804 //
0805 // +--------10---------+-------10---------+---------12----------+
0806 // ¦ ページディレクトリ¦ ページテーブル   ¦ ページ内オフセット  ¦
0807 // ¦   インデックス    ¦ インデックス     ¦                     ¦
0808 // +------------------+-----------------+-----------------------+
0809 // \---- PDX(va) ------/\--- PTX(va) -----/
0810
0811 // ページディレクトリ・インデックス
0812 #define PDX(va)         (((uint)(va) >> PDXSHIFT) & 0x3FF)
0813
0814 // ページテーブル・インデックス
0815 #define PTX(va)         (((uint)(va) >> PTXSHIFT) & 0x3FF)
0816
0817 // インデックスとオフセットから仮想アドレスを構成
0818 #define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT ¦ (t) << PTXSHIFT ¦ (o)))
0819
0820 // ページディレクトリとページテーブルの定数
0821 #define NPDENTRIES      1024    // PGDIRあたりのディレクトリエントリ数
0822 #define NPTENTRIES      1024    // ページテーブルあたりのPTE数
0823 #define PGSIZE          4096    // 1ページにマッピングされるバイト数
0824
0825 #define PGSHIFT         12      // log2(PGSIZE)
0826 #define PTXSHIFT        12      // リニアアドレスにおけるPTXのオフセット
0827 #define PDXSHIFT        22      // リニアアドレスにおけるPDXのオフセット
0828
0829 #define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & □(PGSIZE-1))
0830 #define PGROUNDDOWN(a) (((a)) & □(PGSIZE-1))
0831
0832 // ページテーブル/ディレクトリエントリのフラグ (p.308)
0833 #define PTE_P           0x001   // 0: メモリ上に存在
0834 #define PTE_W           0x002   // 1: 書き込み可能
0835 #define PTE_U           0x004   // 2: ユーザ
0836 #define PTE_PWT         0x008   // 3: ライトスルー
0837 #define PTE_PCD         0x010   // 4: キャッシュ禁止
0838 #define PTE_A           0x020   // 5: アクセス
0839 #define PTE_D           0x040   // 6: ダーティ
0840 #define PTE_PS          0x080   // 7: ページサイズ
0841 #define PTE_MBZ         0x180   // 7-8: ビットは0固定
0842
0843 // ページテーブル/ディレクトリエントリ内のアドレス
0844 #define PTE_ADDR(pte)   ((uint)(pte) & □0xFFF)
0845 #define PTE_FLAGS(pte)  ((uint)(pte) &  0xFFF)
0846
0847 #ifndef __ASSEMBLER__
0848 typedef uint pte_t;
0849
```



```
0850 // タスクステートセグメント形式
0851 struct taskstate {
0852   uint link;         // 旧タスクステートセレクタ
0853   uint esp0;         // 特権レベルが上がった後の
0854   ushort ss0;        //   スタックポインタとセグメントセレクタ
0855   ushort padding1;
0856   uint *esp1;
0857   ushort ss1;
0858   ushort padding2;
0859   uint *esp2;
0860   ushort ss2;
0861   ushort padding3;
0862   void *cr3;         // ページディレクトリのベース
0863   uint *eip;         // 直近のタスクスイッチで保存されたステート
0864   uint eflags;
0865   uint eax;          // さらに保存されたステート ( レジスタ )
0866   uint ecx;
0867   uint edx;
0868   uint ebx;
0869   uint *esp;
0870   uint *ebp;
0871   uint esi;
0872   uint edi;
0873   ushort es;         // さらにさらに保存されたステート ( セグメントセレクタ )
0874   ushort padding4;
0875   ushort cs;
0876   ushort padding5;
0877   ushort ss;
0878   ushort padding6;
0879   ushort ds;
0880   ushort padding7;
0881   ushort fs;
0882   ushort padding8;
0883   ushort gs;
0884   ushort padding9;
0885   ushort ldt;
0886   ushort padding10;
0887   ushort t;          // タスクスイッチを起こしたトラップ
0888   ushort iomb;       // I/Oマップのベースアドレス
0889 };
0890
0891
0892
0893
0894
0895
0896
0897
0898
0899
```

```
0900 // 割り込みとトラップ用のゲートディスクリプタ
0901 struct gatedesc {
0902   uint off_15_0 : 16;   // セグメントオフセットの低位16ビット
0903   uint cs : 16;         // コードセグメントセレクタ
0904   uint args : 5;        // 引数の数、割り込み/トラップゲートでは0
0905   uint rsv1 : 3;        // 予約済（0にするべきだと思う）
0906   uint type : 4;        // タイプ(STS_{TG,IG32,TG32})
0907   uint s : 1;           // 0固定（システム）
0908   uint dpl : 2;         // ディスクリプタ（の新しい）特権レベル
0909   uint p : 1;           // 存在する
0910   uint off_31_16 : 16;  // セグメントオフセットの高位16ビット
0911 };
0912
0913 // 通常の(割り込み/トラップ)ゲートディスクリプタを設定する。
0914 // - istrap: 1 はトラップ（=例外）ゲート、0 は割り込みゲート。
0915 //   割り込みゲートはFL_FLをクリアするが、トラップゲートはFL_IFをいじらない
0916 // - sel: 割り込み/トラップハンドラ用のコードセグメントセレクタ
0917 // - off: 割り込み/トラップハンドラ用のコードセグメント内のオフセット
0918 // - dpl: ディスクリプタ特権レベル -
0919 //        ソフトウェアがint命令を使ってこの割り込み/トラップゲートを
0920 //        明示的に実行するために必要な特権レベル
0921 #define SETGATE(gate, istrap, sel, off, d)                \
0922 {                                                         \
0923   (gate).off_15_0 = (uint)(off) & 0xffff;                \
0924   (gate).cs = (sel);                                      \
0925   (gate).args = 0;                                        \
0926   (gate).rsv1 = 0;                                        \
0927   (gate).type = (istrap) ? STS_TG32 : STS_IG32;           \
0928   (gate).s = 0;                                           \
0929   (gate).dpl = (d);                                       \
0930   (gate).p = 1;                                           \
0931   (gate).off_31_16 = (uint)(off) >> 16;                  \
0932 }
0933
0934 #endif
0935
0936
0937
0938
0939
0940
0941
0942
0943
0944
0945
0946
0947
0948
0949
```



```
0950 // ELF実行ファイルのフォーマット
0951
0952 #define ELF_MAGIC 0x464C457FU  // リトルエンディアンで"\x7FELF"
0953
0954 // ファイルヘッダ
0955 struct elfhdr {
0956   uint magic;  // ELF_MAGICに等しくなければならない
0957   uchar elf[12];
0958   ushort type;
0959   ushort machine;
0960   uint version;
0961   uint entry;
0962   uint phoff;
0963   uint shoff;
0964   uint flags;
0965   ushort ehsize;
0966   ushort phentsize;
0967   ushort phnum;
0968   ushort shentsize;
0969   ushort shnum;
0970   ushort shstrndx;
0971 };
0972
0973 // プログラムセクションヘッダ
0974 struct proghdr {
0975   uint type;
0976   uint off;
0977   uint vaddr;
0978   uint paddr;
0979   uint filesz;
0980   uint memsz;
0981   uint flags;
0982   uint align;
0983 };
0984
0985 // プログラムヘッダのtype値
0986 #define ELF_PROG_LOAD           1
0987
0988 // プログラムヘッダのflags値のフラグビット
0989 #define ELF_PROG_FLAG_EXEC      1
0990 #define ELF_PROG_FLAG_WRITE     2
0991 #define ELF_PROG_FLAG_READ      4
0992
0993
0994
0995
0996
0997
0998
0999
```

```
# xv6のカーネルはこのファイルから実行を開始する。このファイルはカーネルの
# Cコードにリンクされているので、main()などのカーネルシンボルを参照できる。
# ブートブロック(bootasm.Sとbootmain.c)はこのファイルのentryにジャンプする。

# マルチブートヘッダ。GUN Grubなどのマルチブート・ブートローダ用。
# http://www.gnu.org/software/grub/manual/multiboot/multiboot.html
#
# GRUB 2を使って、Linuxファイルシステムに格納したファイルからxv6を
# ブートできる。kernelまたはkernelmemfsを/bootにコピーし、
# 以下をメニューエントリに追加する。
#
# 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"

# マルチブートヘッダ。マルチブートローダへ向けるためのデータ。
.p2align 2
.text
.globl multiboot_header
multiboot_header:
  #define magic 0x1badb002
  #define flags 0
  .long magic
  .long flags
  .long (-magic-flags)

# 規約により、_startシンボルにはELFエントリポイントを指定する。
# まだ仮想メモリを設定していないので、エントリポイントは
# 'entry'の物理アドレスである。
.globl _start
_start = V2P_WO(entry)

# ブートプロセッサのxv6に入る。ページングは無効。
.globl entry
entry:
  # ページ単位が4MBになるようにページサイズ拡張をセットする
  movl    %cr4, %eax
  orl     $(CR4_PSE), %eax
  movl    %eax, %cr4
  # ページディレクトリをセットする
```



```
  movl    $(V2P_WO(entrypgdir)), %eax
  movl    %eax, %cr3
  # ページングを有効にする
  movl    %cr0, %eax
  orl     $(CR0_PG¦CR0_WP), %eax
  movl    %eax, %cr0

  # スタックポインタを設定する
  movl $(stack + KSTACKSIZE), %esp

  # main()にジャンプして、高位アドレスの実行コードにスイッチする。
  # ここでは間接呼び出しが必要である。
  # なぜなら、直接ジャンプにすると
  # アセンブラがPC相対の命令を生成するからである。
  mov $main, %eax
  jmp *%eax

.comm stack, KSTACKSIZE
```

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

# 非ブートCPU( "AP" )はブートCPUからのSTARTUP IPIに反応して開始される。
# "Multi-Processor Specification"のB.4.2節によると、
# APは、CS:IPにXY00:0000がセットされたリアルモードで開始する。
# ここで、XYはSTARTUPで送信される8ビットの値である。
# したがって、このコードは4096バイト境界から開始しなければならない。
#
# このコードはDSに0をセットするので、
# 低位2^16(64K)バイトのアドレスにいなければならない。
#
# (main.cの )startothersは一度に1つSTARTUPを送信する。
# startothersはこのコード(start)を0x7000にコピーし、
# コアごとに新規に割り当てられるスタックのアドレスを(start-4)に、
# ジャンプ先のアドレス(mpenter)を(start-8)に、
# entrypgdirの物理アドレスを(start-12)に置く。
#
# このコードはbootasm.Sとentry.Sの要素を結合する

.code16
.globl start
start:
  cli

  # データセグメントレジスタDS, ES, SSを0クリアする。
  xorw    %ax,%ax
  movw    %ax,%ds
  movw    %ax,%es
  movw    %ax,%ss

  # リアルモードからプロテクトモードに切り替える。切り替えの際に
  # 実効メモリマップが変わらないように、仮想アドレスをそのまま
  # 物理アドレスにマッピングするブートストラップGDTを使用する。
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE, %eax
  movl    %eax, %cr0

  # %csと%eipを再ロードするためのロングジャンプを利用して、32ビット
  # プロテクトモードへの移行を完了させる。セグメントディスクリプタは
  # 無変換に設定されているので、マッピングは依然として恒等マッピングである。
  ljmpl    $(SEG_KCODE<<3), $(start32)
```



```
.code32  # ここからは32ビットコードを生成するようアセンブラに伝える
start32:
  # プロテクトモードのデータセグメントレジスタを設定する
  movw    $(SEG_KDATA<<3), %ax    # データセグメントセレクタ(0x10)
  movw    %ax, %ds                # -> DS: データセグメント
  movw    %ax, %es                # -> ES: エクストラセグメント
  movw    %ax, %ss                # -> SS: スタックセグメント
  movw    $0, %ax                 # 未使用のセグメントには0をセット
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS

  # ページサイズが4MBになるようにPSE(ページサイズ拡張)ビットをセットする
  movl    %cr4, %eax
  orl     $(CR4_PSE), %eax
  movl    %eax, %cr4
  # 初期ページテーブルとしてentrypgdirを使用する
  movl    (start-12), %eax
  movl    %eax, %cr3
  # ページングを有効にする
  movl    %cr0, %eax
  orl     $(CR0_PE¦CR0_PG¦CR0_WP), %eax
  movl    %eax, %cr0

  # startothers()で割り当てられたスタックに切り替える
  movl    (start-4), %esp
  # mpenter()を呼び出す
  call         *(start-8)

  movw    $0x8a00, %ax
  movw    %ax, %dx
  outw    %ax, %dx
  movw    $0x8ae0, %ax
  outw    %ax, %dx
spin:
  jmp     spin

.p2align 2
gdt:
  SEG_NULLASM
  SEG_ASM(STA_X¦STA_R, 0, 0xffffffff)
  SEG_ASM(STA_W, 0, 0xffffffff)


gdtdesc:
  .word   (gdtdesc - gdt - 1)
  .long   gdt
```

```
1200 #include "types.h"
1201 #include "defs.h"
1202 #include "param.h"
1203 #include "memlayout.h"
1204 #include "mmu.h"
1205 #include "proc.h"
1206 #include "x86.h"
1207
1208 static void startothers(void);
1209 static void mpmain(void)  __attribute__((noreturn));
1210 extern pde_t *kpgdir;
1211 extern char end[]; // カーネルをELFファイルからロードした後の最初のアドレス
1212
1213 // ブートストラップププロセッサはここからCコードの実行を開始する。
1214 // 実際のスタックを割り当て、それに切り替える。まず、メモリアロケータの
1215 // 動作に必要な各種設定を行う
1216 int
1217 main(void)
1218 {
1219   kinit1(end, P2V(4*1024*1024)); // 物理ページアロケータ
1220   kvmalloc();      // カーネルページテーブル
1221   mpinit();        // 他のプロセッサの検出
1222   lapicinit();     // 割り込みコントローラ
1223   seginit();       // セグメントデスクリプタ
1224   picinit();       // picの無効化
1225   ioapicinit();    // もう1つの割り込みコントローラ
1226   consoleinit();   // コンソールハードウェア
1227   uartinit();      // シリアルポート
1228   pinit();         // プロセステーブル
1229   tvinit();        // トラップベクタ
1230   binit();         // バッファキャッシュ
1231   fileinit();      // ファイルテーブル
1232   ideinit();       // ディスク
1233   startothers();   // 他のプロセッサの開始
1234   kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // startothers()の後でなければならない
1235   userinit();      // 最初のユーザプロセス
1236   mpmain();        // このプロセッサの設定を終了
1237 }
1238
1239 // 他のCPUはentryother.Sからここにジャンプする
1240 static void
1241 mpenter(void)
1242 {
1243   switchkvm();
1244   seginit();
1245   lapicinit();
1246   mpmain();
1247 }
1248
1249
```



```
1250 // 共通のCPU設定コード
1251 static void
1252 mpmain(void)
1253 {
1254   cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
1255   idtinit();       // idtレジスタのロード
1256   xchg(&(mycpu()->started), 1); // startothers()にこのCPUの起動を伝える
1257   scheduler();     // プロセスの実行を開始する
1258 }
1259
1260 pde_t entrypgdir[];  // entry.S用に
1261
1262 // 非ブート(AP)プロセッサを開始する
1263 static void
1264 startothers(void)
1265 {
1266   extern uchar _binary_entryother_start[], _binary_entryother_size[];
1267   uchar *code;
1268   struct cpu *c;
1269   char *stack;
1270
1271   // エントリコードを0x7000の未使用メモリに書き込む
1272   // リンカはentryother.Sのイメージを_binary_entryother_startに
1273   // 置いている。
1274   code = P2V(0x7000);
1275   memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);
1276
1277   for(c = cpus; c < cpus+ncpu; c++){
1278     if(c == mycpu())  // ブートCPUは開始済み
1279       continue;
1280
1281     // どのスタックを使うのか、どこにenterするのか、どのpgdirを使うのかを
1282     // entryother.Sに伝える。APプロセッサは低位アドレスで実行しているので
1283     // kpgdirはまだ使えない。そのため、APにもentrypgdirを使用する。
1284     stack = kalloc();
1285     *(void**)(code-4) = stack + KSTACKSIZE;
1286     *(void**)(code-8) = mpenter;
1287     *(int**)(code-12) = (void *) V2P(entrypgdir);
1288
1289     lapicstartap(c->apicid, V2P(code));
1290
1291     // cpuがmpmain()を終了するのを待機する
1292     while(c->started == 0)
1293       ;
1294   }
1295 }
1296
1297
1298
1299
```

```
// entry.S と entryother.Sで使用するブートページテーブル.
// ページディレクトリ ( とページエントリ ) はページ境界から開始しなければ
// ならない。そのため __aligned__ 属性を指定する。
// ページディレクトリエントリの PTE_PS はページサイズを4Mバイトにする。

__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
  // 仮想アドレス [0, 4MB) を物理アドレス [0, 4MB) にマッピング
  [0] = (0) ¦ PTE_P ¦ PTE_W ¦ PTE_PS,
  // 仮想アドレス [KERNBASE, KERNBASE+4MB) を物理アドレス [0, 4MB)に
  [KERNBASE>>PDXSHIFT] = (0) ¦ PTE_P ¦ PTE_W ¦ PTE_PS,
};
```



```
// Blank page.
```

```
// Blank page.
```



```
// Blank page.
```

```
1500 // 相互排他ロック
1501 struct spinlock {
1502   uint locked;       // このロックは獲得済みか?
1503
1504   // デバック用:
1505   char *name;        // ロックの名前
1506   struct cpu *cpu;   // ロックを保持しているCPU
1507   uint pcs[10];      // このロックをロックしたコールスタック
1508                      // ( プログラムカウンタの配列 )
1509 };
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
```



```
1550 // 相互排他スピンロック。
1551
1552 #include "types.h"
1553 #include "defs.h"
1554 #include "param.h"
1555 #include "x86.h"
1556 #include "memlayout.h"
1557 #include "mmu.h"
1558 #include "proc.h"
1559 #include "spinlock.h"
1560
1561 void
1562 initlock(struct spinlock *lk, char *name)
1563 {
1564   lk->name = name;
1565   lk->locked = 0;
1566   lk->cpu = 0;
1567 }
1568
1569 // ロックを獲得する。
1570 // ロックが獲得されるまでループ ( スピン ) する。
1571 // ロックを長時間保持すると、他のCPUがロックの
1572 // 獲得のためにスピンして時間を浪費する可能性がある。
1573 void
1574 acquire(struct spinlock *lk)
1575 {
1576   pushcli(); // ヘッドロックを避けるために割り込みを禁止する。
1577   if(holding(lk))
1578     panic("acquire");
1579
1580   // xchgはアトミックである。
1581   while(xchg(&lk->locked, 1) != 0)
1582     ;
1583
1584   // クリティカルセクションのメモリ参照がロックの獲得後に行われるように、
1585   // この点を超えるロード/ストアの移動をしないように
1586   // Cコンパイラとプロセッサに指示する。( メモリバリア )
1587   __sync_synchronize();
1588
1589   // デバッグ用にロック獲得に関する情報を記録する。
1590   lk->cpu = mycpu();
1591   getcallerpcs(&lk, lk->pcs);
1592 }
1593
1594
1595
1596
1597
1598
1599
```

```
1600 // ロックを解放する。
1601 void
1602 release(struct spinlock *lk)
1603 {
1604   if(!holding(lk))
1605     panic("release");
1606
1607   lk->pcs[0] = 0;
1608   lk->cpu = 0;
1609
1610   // クリティカルセクションのすべてのストアがロックが解放される前に、
1611   // 他のコアから見られるようにするために、この点を超えるロード/ストアの
1612   // 移動をしないように、Cコンパイラとプロセッサに指ぇする。
1613   // Cコンパイラとプロセッサは共にロードとストアを再配置する可能性がある。
1614   // __sync_synchronize() は両者にそれをしないように指ぇする。
1615   __sync_synchronize();
1616
1617   // ロックの解放は、lk->locked = 0 に相当する。
1618   // このコードにCの代入は使えない。Cの代入はアトミックでは
1619   // ないからである。実際のOSはCのアトミック関数をここに使うだろう。
1620   asm volatile("movl $0, %0" : "+m" (lk->locked) : );
1621
1622   popcli();
1623 }
1624
1625 // %ebpチェインをたどり、pcs[]に現在のコーススタックを記録する。
1626 void
1627 getcallerpcs(void *v, uint pcs[])
1628 {
1629   uint *ebp;
1630   int i;
1631
1632   ebp = (uint*)v - 2;
1633   for(i = 0; i < 10; i++){
1634     if(ebp == 0 ¦¦ ebp < (uint*)KERNBASE ¦¦ ebp == (uint*)0xffffffff)
1635       break;
1636     pcs[i] = ebp[1];     // 保存されていた%eip
1637     ebp = (uint*)ebp[0]; // 保存されていた%ebp
1638   }
1639   for(; i < 10; i++)
1640     pcs[i] = 0;
1641 }
1642
1643 // このCPUがロックを保持しているかチェックする。
1644 int
1645 holding(struct spinlock *lock)
1646 {
1647   return lock->locked && lock->cpu == mycpu();
1648 }
1649
```



```
1650 // Pushcli/popcliは両者の数が一致することを除いて、cli/stiと同じである:
1651 // 2回のpushcliを取り消すには2回のpopcliが必要である。また、割り込みが
1652 // 無効であれば、pushcliとpopcliは割り込みを無効のままにする。
1653
1654 void
1655 pushcli(void)
1656 {
1657   int eflags;
1658
1659   eflags = readeflags();
1660   cli();
1661   if(mycpu()->ncli == 0)
1662     mycpu()->intena = eflags & FL_IF;
1663   mycpu()->ncli += 1;
1664 }
1665
1666 void
1667 popcli(void)
1668 {
1669   if(readeflags()&FL_IF)
1670     panic("popcli - interruptible");
1671   if(--mycpu()->ncli < 0)
1672     panic("popcli");
1673   if(mycpu()->ncli == 0 && mycpu()->intena)
1674     sti();
1675 }
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
```

```
1700 #include "param.h"
1701 #include "types.h"
1702 #include "defs.h"
1703 #include "x86.h"
1704 #include "memlayout.h"
1705 #include "mmu.h"
1706 #include "proc.h"
1707 #include "elf.h"
1708
1709 extern char data[];  // kernel.ldで定義
1710 pde_t *kpgdir;  // scheduler()で使用される
1711
1712 // CPUのカーネルセグメントディスクリプタを設定する。
1713 // 各CPUのentryで1回ずつ実行される。
1714 void
1715 seginit(void)
1716 {
1717   struct cpu *c;
1718
1719   // 恒等マッピングを使用して「論理」アドレスを仮想アドレスにマッピングする。
1720   // カーネル用とユーザ用のコードディスクリプタの共用はできない。
1721   // なぜなら、共用する場合はDPL_USRを持たねばならないが、CPUはCPL=0から
1722   // DPL=3への割り込みを禁止しているからである。
1723   c = &cpus[cpuid()];
1724   c->gdt[SEG_KCODE] = SEG(STA_X¦STA_R, 0, 0xffffffff, 0);
1725   c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
1726   c->gdt[SEG_UCODE] = SEG(STA_X¦STA_R, 0, 0xffffffff, DPL_USER);
1727   c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
1728   lgdt(c->gdt, sizeof(c->gdt));
1729 }
1730
1731 // 仮想アドレスvaに対応するページテーブルpgdirのPTEのアドレスを返す。
1732 // alloc != 0 の場合は、
1733 // 必要なページテーブルページを作成する。
1734 static pte_t *
1735 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1736 {
1737   pde_t *pde;
1738   pte_t *pgtab;
1739
1740   pde = &pgdir[PDX(va)];
1741   if(*pde & PTE_P){
1742     pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
1743   } else {
1744     if(!alloc ¦¦ (pgtab = (pte_t*)kalloc()) == 0) // 割当不要または失敗の場合は0を返す
1745       return 0;
1746     // すべてのページテーブルエントリのPTE_Pビットを0にする
1747     memset(pgtab, 0, PGSIZE);
1748     // 通常、ここでのパーミッションは上書きされるが、
1749     // 必要であれば、ページテーブルエントリのパーミションでさらに制限する
```



```
1750     // こともできる。
1751     *pde = V2P(pgtab) ¦ PTE_P ¦ PTE_W ¦ PTE_U;
1752   }
1753   return &pgtab[PTX(va)];
1754 }
1755
1756 // vaから始まる仮想アドレスがpaから始まる物理アドレスを参照するように
1757 // するためのPTE（ページテーブルエントリ）を作成する。
1758 // vaとサイズはページ境界にない可能性がある。
1759 static int
1760 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1761 {
1762   char *a, *last;
1763   pte_t *pte;
1764
1765   a = (char*)PGROUNDDOWN((uint)va);
1766   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1767   for(;;){
1768     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1769       return -1;
1770     if(*pte & PTE_P)
1771       panic("remap");
1772     *pte = pa ¦ perm ¦ PTE_P;
1773     if(a == last)
1774       break;
1775     a += PGSIZE;
1776     pa += PGSIZE;
1777   }
1778   return 0;
1779 }
1780
1781 // ページテーブルはプロセスごとに1つあり、さらに1つ、CPUがプロセスを
1782 // 1つも実行していない時に使用するページテーブルがある(kpgdir)。
1783 // システムコールと割り込み処理の際、カーネルはカレントプロセスの
1784 // ページテーブルを使用する。ページ保護ビットがユーザコードによる
1785 // カーネルマッピングの使用を禁止する。
1786 //
1787 // setupkvm() と exec() はすべてのページテーブルを次のように設定する:
1788 //
1789 //   0..KERNBASE: ユーザメモリ (text+data+stack+heap)
1790 //                カーネルにより割り当てられる物理メモリにマッピングされる
1791 //   KERNBASE..KERNBASE+EXTMEM: 0..EXTMEM にマッピングされる(I/O 空間用)
1792 //   KERNBASE+EXTMEM..data: EXTMEM..V2P(data)にマッピングされる
1793 //                カーネルの命令コードと読み込み専用データ用
1794 //   data..KERNBASE+PHYSTOP: V2P(data)..PHYSTOPにマッピングされる
1795 //                読み書きデータと空き物理メモリ
1796 //   0xfe000000..0: そのままマッピングされる（ioapicなどのデバイス）
1797 //
1798 // カーネルは自身のヒープ用とユーザメモリ用の物理メモリを
1799 // V2P(end)から物理メモリ上限(PHYSTOP)の間に割り当てる。
```

```
// （ end..P2V(PHYSTOP)は直接アドレスすることができる ）

// このテーブルはカーネルのマッピングを定義する。これは全プロセスの
// ページテーブルに現れる
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {
 { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O空間
 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // カーネルのtext+rodata
 { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // カーネルのdata+memory
 { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // デバイス
};

// ページテーブルのカーネル部分を設定する
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)    // ページテーブルの割り当て
    return 0;
  memset(pgdir, 0, PGSIZE);              // ページテーブルを0詰め
  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) {
      freevm(pgdir);
      return 0;
    }
  return pgdir;
}

// スケジューラプロセスが使用するカーネルアドレス空間用の
// ページテーブルをマシンに1つ割り当てる
void
kvmalloc(void)
{
  kpgdir = setupkvm();
  switchkvm();
}
```



```
// 実行中のプロセスがない場合は、ハードウェアのページテーブルレジスタを
// カーネル専用のページテーブルに切り替える
void
switchkvm(void)
{
  lcr3(V2P(kpgdir));   // カーネルページテーブルに切り替える
}

// TSSとハードウェアページテーブルをプロセスpのものに切り替える
void
switchuvm(struct proc *p)
{
  if(p == 0)
    panic("switchuvm: no process");
  if(p->kstack == 0)
    panic("switchuvm: no kstack");
  if(p->pgdir == 0)
    panic("switchuvm: no pgdir");

  pushcli();
  mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
                                sizeof(mycpu()->ts)-1, 0);
  mycpu()->gdt[SEG_TSS].s = 0;       // TSSディスクリプタ
  mycpu()->ts.ss0 = SEG_KDATA << 3;
  mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
  // eflagsにおけるIOPL=0の設定、*かつ*、iombへのtssセグメントのリミット値を
  // 超える値の設定により、ユーザ空間からのI/O命令（ inbとoutbなど ）を禁止する。
  mycpu()->ts.iomb = (ushort) 0xFFFF;
  ltr(SEG_TSS << 3);
  lcr3(V2P(p->pgdir));  // プロセスのアドレス空間に切り替える
  popcli();
}

// initcodeをpgdirのアドレス0にロードする。
// sz は1ページ未満でなければならない。
void
inituvm(pde_t *pgdir, char *init, uint sz)
{
  char *mem;

  if(sz >= PGSIZE)
    panic("inituvm: more than a page");
  mem = kalloc();  // 4096byte = 1 pageを割り当て
  memset(mem, 0, PGSIZE);
  mappages(pgdir, 0, PGSIZE, V2P(mem), PTE_W¦PTE_U);
  memmove(mem, init, sz);
}
```

```
1900 // プログラムセグメントをpgdirにロードする。addrはページ境界になければならない。
1901 // また、addrからaddr+szのページはマッピング済みでなければならない。
1902 int
1903 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
1904 {
1905   uint i, pa, n;
1906   pte_t *pte;
1907
1908   if((uint) addr % PGSIZE != 0)
1909     panic("loaduvm: addr must be page aligned");
1910   for(i = 0; i < sz; i += PGSIZE){
1911     if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
1912       panic("loaduvm: address should exist");
1913     pa = PTE_ADDR(*pte);
1914     if(sz - i < PGSIZE)
1915       n = sz - i;
1916     else
1917       n = PGSIZE;
1918     if(readi(ip, P2V(pa), offset+i, n) != n)
1919       return -1;
1920   }
1921   return 0;
1922 }
1923
1924 // プロセスをoldszからnewszに拡張するためにページテーブルと物理メモリを割り当てる。
1925 // サイズはページ境界になくても良い。新しいサイズを返す。エラーの場合は0を返す。
1926 int
1927 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1928 {
1929   char *mem;
1930   uint a;
1931
1932   if(newsz >= KERNBASE)
1933     return 0;
1934   if(newsz < oldsz)
1935     return oldsz;
1936
1937   a = PGROUNDUP(oldsz);
1938   for(; a < newsz; a += PGSIZE){
1939     mem = kalloc();
1940     if(mem == 0){
1941       cprintf("allocuvm out of memory\n");
1942       deallocuvm(pgdir, newsz, oldsz);
1943       return 0;
1944     }
1945     memset(mem, 0, PGSIZE);
1946     if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W¦PTE_U) < 0){
1947       cprintf("allocuvm out of memory (2)\n");
1948       deallocuvm(pgdir, newsz, oldsz);
1949       kfree(mem);
```



```
1950       return 0;
1951     }
1952   }
1953   return newsz;
1954 }
1955
1956 // プロセスサイズをoldszからnewszにするためにユーザページの割り当てを解除する。
1957 // oldszとnewszはページ境界になくても良い。また、newszはoldszより小さくなくても良い。
1958 // oldszは実際のプロセスサイズより大きくても良い
1959 // 新しいプロセスサイズを返す
1960 int
1961 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1962 {
1963   pte_t *pte;
1964   uint a, pa;
1965
1966   if(newsz >= oldsz)
1967     return oldsz;
1968
1969   a = PGROUNDUP(newsz);
1970   for(; a  < oldsz; a += PGSIZE){
1971     pte = walkpgdir(pgdir, (char*)a, 0);
1972     if(!pte)
1973       a = PGADDR(PDX(a) + 1, 0, 0) - PGSIZE;
1974     else if((*pte & PTE_P) != 0){
1975       pa = PTE_ADDR(*pte);
1976       if(pa == 0)
1977         panic("kfree");
1978       char *v = P2V(pa);
1979       kfree(v);
1980       *pte = 0;
1981     }
1982   }
1983   return newsz;
1984 }
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
```

```
2000 // ページテーブルとユーザ部分のすべての物理メモリを
2001 // 解放する。
2002 void
2003 freevm(pde_t *pgdir)
2004 {
2005   uint i;
2006
2007   if(pgdir == 0)
2008     panic("freevm: no pgdir");
2009   deallocuvm(pgdir, KERNBASE, 0);
2010   for(i = 0; i < NPDENTRIES; i++){
2011     if(pgdir[i] & PTE_P){
2012       char * v = P2V(PTE_ADDR(pgdir[i]));
2013       kfree(v);
2014     }
2015   }
2016   kfree((char*)pgdir);
2017 }
2018
2019 // ページのPTE_Uをクリアする。ユーザスタック直下に
2020 // アクセス不能なページを作成するのに使用する
2021 void
2022 clearpteu(pde_t *pgdir, char *uva)
2023 {
2024   pte_t *pte;
2025
2026   pte = walkpgdir(pgdir, uva, 0);
2027   if(pte == 0)
2028     panic("clearpteu");
2029   *pte &= ~PTE_U;
2030 }
2031
2032 // 親プロセスのページテーブルを与え、子プロセス用に
2033 // それのコピーを作成する。
2034 pde_t*
2035 copyuvm(pde_t *pgdir, uint sz)
2036 {
2037   pde_t *d;
2038   pte_t *pte;
2039   uint pa, i, flags;
2040   char *mem;
2041
2042   if((d = setupkvm()) == 0)
2043     return 0;
2044   for(i = 0; i < sz; i += PGSIZE){
2045     if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
2046       panic("copyuvm: pte should exist");
2047     if(!(*pte & PTE_P))
2048       panic("copyuvm: page not present");
2049     pa = PTE_ADDR(*pte);
```



```
2050     flags = PTE_FLAGS(*pte);
2051     if((mem = kalloc()) == 0)
2052       goto bad;
2053     memmove(mem, (char*)P2V(pa), PGSIZE);
2054     if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0)
2055       goto bad;
2056   }
2057   return d;
2058
2059 bad:
2060   freevm(d);
2061   return 0;
2062 }
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
```

```
2100 // ユーザ仮想アドレスをカーネルアドレスにマッピングする。
2101 char*
2102 uva2ka(pde_t *pgdir, char *uva)
2103 {
2104   pte_t *pte;
2105
2106   pte = walkpgdir(pgdir, uva, 0);
2107   if((*pte & PTE_P) == 0)
2108     return 0;
2109   if((*pte & PTE_U) == 0)
2110     return 0;
2111   return (char*)P2V(PTE_ADDR(*pte));
2112 }
2113
2114 // pからページテーブル pgdirのユーザアドレス va へ lenバイトコピーする。
2115 // pgdirがカレントページテーブルでない場合に最も役に立つ。
2116 // uva2ka はこの関数がPTF_Uページでのみ動作することを保証する。
2117 int
2118 copyout(pde_t *pgdir, uint va, void *p, uint len)
2119 {
2120   char *buf, *pa0;
2121   uint n, va0;
2122
2123   buf = (char*)p;
2124   while(len > 0){
2125     va0 = (uint)PGROUNDDOWN(va);
2126     pa0 = uva2ka(pgdir, (char*)va0);
2127     if(pa0 == 0)
2128       return -1;
2129     n = PGSIZE - (va - va0);
2130     if(n > len)
2131       n = len;
2132     memmove(pa0 + (va - va0), buf, n);
2133     len -= n;
2134     buf += n;
2135     va = va0 + PGSIZE;
2136   }
2137   return 0;
2138 }
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
```



```
2150 // Blank page.
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
```

```
2200 // Blank page.
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
```



```
2250 // Blank page.
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
```

```
2300 // CPU毎のステート
2301 struct cpu {
2302   uchar apicid;                // ローカルAPIC ID
2303   struct context *scheduler;   // スケジューラに入るにはここにswtch()
2304   struct taskstate ts;         // 割り込み用のスタックを探すためにx86が使用
2305   struct segdesc gdt[NSEGS];   // x86グローバルディスクリプタテーブル
2306   volatile uint started;       // CPUは開始しているか?
2307   int ncli;                    // pushcliネストの深さ
2308   int intena;                  // pushcliの前に割り込みは有効だったか?
2309   struct proc *proc;           // このCPUで実行中のプロセス。なければnull
2310 };
2311
2312 extern struct cpu cpus[NCPU];
2313 extern int ncpu;
2314
2315
2316 // カーネルコンテキストスイッチで保存されるレジスタ
2317 // すべてのセグメントレジスタ(%csなど)はカーネルコンテキストを
2318 // 通じて不変であるため保存する必要はない。
2319 // %eax, %ecx, %edxは、x86の規約で呼び出し元が保存するので、
2320 // 保存する必要はない。
2321 // コンテキストは自身が記述するスタックの底に格納される。
2322 // すなわち、スタックポインタはコンテキストのアドレスである。
2323 // コンテキストのレイアウトは、switch.Sのコメント"スタックを切り替える"に
2324 // 書かれているスタックのレイアウトに一致する。スイッチはeipを明示的には
2325 // 保存しないが、スタック上にあり、allocproc()がそれを処理する。
2326 struct context {
2327   uint edi;
2328   uint esi;
2329   uint ebx;
2330   uint ebp;
2331   uint eip;
2332 };
2333
2334 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
2335
2336 // プロセスごとのステート
2337 struct proc {
2338   uint sz;                     // プロセスメモリのサイズ(単位はバイト)
2339   pde_t* pgdir;                // ページテーブル
2340   char *kstack;                // このプロセスのカーネルスタックの底
2341   enum procstate state;        // プロセスの状態
2342   int pid;                     // プロセスID
2343   struct proc *parent;         // 親プロセス
2344   struct trapframe *tf;        // 現在のsyscallのトラップフレーム
2345   struct context *context;     // プロセスの実行するためにここにswtch()
2346   void *chan;                  // 非ゼロの場合、chanでスリープ中
2347   int killed;                  // 非ゼロの場合、キルされた
2348   struct file *ofile[NOFILE];  // オープンしたファイル
2349   struct inode *cwd;           // カレントディレクトリ
```



```
2350   char name[16];               // プロセス名(デバッグ用)
2351 };
2352
2353 // プロセスメモリは、低位アドレスから次のように、連続的に配置される。
2354 //   テキスト
2355 //   オリジナルのデータとbss
2356 //   固定サイズのスタック
2357 //   拡張可能なヒープ
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
```

```
2400 #include "types.h"
2401 #include "defs.h"
2402 #include "param.h"
2403 #include "memlayout.h"
2404 #include "mmu.h"
2405 #include "x86.h"
2406 #include "proc.h"
2407 #include "spinlock.h"
2408
2409 struct {
2410     struct spinlock lock;
2411     struct proc proc [NPROC];
2412 } ptable;
2413
2414 static struct proc * initproc;
2415
2416 int nextpid = 1;
2417 extern void forkret(void);
2418 extern void trapret(void);
2419
2420 static void wakeup1( void * chan );
2421
2422 void
2423 pinit(void)
2424 {
2425   initlock( &ptable.lock, "ptable" );
2426 }
2427
2428 // 割り込みを禁止してから呼び出す必要がある
2429 int
2430 cpuid() {
2431   return mycpu()-cpus;
2432 }
2433
2434 // lapicidの読み込みとループ実行の間に呼び出し側が再スケジュール
2435 // されないように、割り込みを禁止してから呼び出す必要がある
2436 struct cpu*
2437 mycpu(void)
2438 {
2439   int apicid, i;
2440
2441   if(readeflags()&FL_IF)
2442     panic("mycpu called with interrupts enabled\n");
2443
2444   apicid = lapicid();
2445   // APIC IDは連続であるとは限らない。リバースマップを持つか
2446   // &cpus[i]を格納するレジスタを用意するべきだろう。
2447   for (i = 0; i < ncpu; ++i) {
2448     if (cpus[i].apicid == apicid)
2449       return &cpus[i];
```



```
2450   }
2451   panic("unknown apicid\n");
2452 }
2453
2454 // cpu構造体からprocを読み込む間に再スケジュールされないように、
2455 // 割り込みを禁止する
2456 struct proc*
2457 myproc(void) {
2458   struct cpu *c;
2459   struct proc *p;
2460   pushcli();
2461   c = mycpu();
2462   p = c->proc;
2463   popcli();
2464   return p;
2465 }
2466
2467
2468 // プロセステーブルから状態がUNUSEDのprocを探す。
2469 // 見つかったら、状態をEMBRYOに変更し、カーネルで実行するために
2470 // 必要な状態の初期化を行う。
2471 // 見つからなかった場合は、0を返す。
2472 static struct proc*
2473 allocproc(void)
2474 {
2475   struct proc *p;
2476   char *sp;
2477
2478   acquire(&ptable.lock);
2479
2480   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2481     if(p->state == UNUSED)
2482       goto found;
2483
2484   release(&ptable.lock);
2485   return 0;
2486
2487 found:
2488   p->state = EMBRYO;
2489   p->pid = nextpid++;
2490
2491   release(&ptable.lock);
2492
2493   // カーネルスタックを割り当てる。
2494   if((p->kstack = kalloc()) == 0){
2495     p->state = UNUSED;
2496     return 0;
2497   }
2498   sp = p->kstack + KSTACKSIZE;
2499
```

```
2500   // トラップフレーム用のスペースを確保する。
2501   sp -= sizeof *p->tf;
2502   p->tf = (struct trapframe*)sp;
2503
2504   // forkretから実行を開始するために新たなコンテキストを設定する。
2505   // forkretからはtrapretに復帰する。
2506   sp -= 4;
2507   *(uint*)sp = (uint)trapret;
2508
2509   sp -= sizeof *p->context;
2510   p->context = (struct context*)sp;
2511   memset(p->context, 0, sizeof *p->context);
2512   p->context->eip = (uint)forkret;
2513
2514   return p;
2515 }
2516
2517
2518 // 最初のユーザプロセスを設定する。
2519 void
2520 userinit(void)
2521 {
2522   struct proc *p;
2523   extern char _binary_initcode_start[], _binary_initcode_size[];
2524
2525   p = allocproc();
2526
2527   initproc = p;
2528   if((p->pgdir = setupkvm()) == 0)
2529     panic("userinit: out of memory?");
2530   inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
2531   p->sz = PGSIZE;
2532   memset(p->tf, 0, sizeof(*p->tf));
2533   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2534   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2535   p->tf->es = p->tf->ds;
2536   p->tf->ss = p->tf->ds;
2537   p->tf->eflags = FL_IF;
2538   p->tf->esp = PGSIZE;
2539   p->tf->eip = 0;  // initcode.Sの先頭
2540
2541   safestrcpy(p->name, "initcode", sizeof(p->name));
2542   p->cwd = namei("/");
2543
2544   // このp->stateへの代入により、他のコアがこのプロセスを実行
2545   // できるようになる。acquireは上の書き込みを可視化する。
2546   // また、代入はアトミックではない可能性もあるので、
2547   // ロックが必要である。
2548   acquire(&ptable.lock);
2549
```



```
2550   p->state = RUNNABLE;
2551
2552   release(&ptable.lock);
2553 }
2554
2555 // カレントプロセスのメモリをnバイト増減する。
2556 // 成功した場合は0、失敗した場合は-1を返す。
2557 int
2558 growproc(int n)
2559 {
2560   uint sz;
2561   struct proc *curproc = myproc();
2562
2563   sz = curproc->sz;
2564   if(n > 0){
2565     if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
2566       return -1;
2567   } else if(n < 0){
2568     if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
2569       return -1;
2570   }
2571   curproc->sz = sz;
2572   switchuvm(curproc);
2573   return 0;
2574 }
2575
2576 // pを親としてコピーして新しいプロセスを作成する。
2577 // システムコールから復帰したかのような復帰用のスタックを構成する。
2578 // 呼び出し側は返されたprocのstateをRUNNABLEにセットしなければならない。( Obsolute: copyproc
2579 int
2580 fork(void)
2581 {
2582   int i, pid;
2583   struct proc *np;
2584   struct proc *curproc = myproc();
2585
2586   // プロセスを割り当てる。
2587   if((np = allocproc()) == 0){
2588     return -1;
2589   }
2590
2591   // curprocからプロセスの状態をコピーする。
2592   if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
2593     kfree(np->kstack);
2594     np->kstack = 0;
2595     np->state = UNUSED;
2596     return -1;
2597   }
2598   np->sz = curproc->sz;
2599   np->parent = curproc;
```

```
2600   *np->tf = *curproc->tf;
2601
2602   // 子プロセスではforkが0を返すように%eaxをクリアする。
2603   np->tf->eax = 0;
2604
2605   for(i = 0; i < NOFILE; i++)
2606     if(curproc->ofile[i])
2607       np->ofile[i] = filedup(curproc->ofile[i]);
2608   np->cwd = idup(curproc->cwd);
2609
2610   safestrcpy(np->name, curproc->name, sizeof(curproc->name));
2611
2612   pid = np->pid;
2613
2614   acquire(&ptable.lock);
2615
2616   np->state = RUNNABLE;
2617
2618   release(&ptable.lock);
2619
2620   return pid;
2621 }
2622
2623 // カレントプロセスを終了する。復帰しない。
2624 // 終了したプロセスは、親プロセスがwait()を呼び出してそれが終了したことを
2625 // 知るまで、zombie状態で残る。
2626 void
2627 exit(void)
2628 {
2629   struct proc *curproc = myproc();
2630   struct proc *p;
2631   int fd;
2632
2633   if(curproc == initproc)
2634     panic("init exiting");
2635
2636   // 開いていたファイルをすべて閉じる。
2637   for(fd = 0; fd < NOFILE; fd++){
2638     if(curproc->ofile[fd]){
2639       fileclose(curproc->ofile[fd]);
2640       curproc->ofile[fd] = 0;
2641     }
2642   }
2643
2644   begin_op();
2645   iput(curproc->cwd);
2646   end_op();
2647   curproc->cwd = 0;
2648
2649   acquire(&ptable.lock);
```



```
2650   // 親プロセスはwait()でスリープしている可能性がある。
2651   wakeup1(curproc->parent);
2652
2653   // 子プロセスは見捨ててinitに渡す。
2654   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2655     if(p->parent == curproc){
2656       p->parent = initproc;
2657       if(p->state == ZOMBIE)
2658         wakeup1(initproc);
2659     }
2660   }
2661
2662   // スケジューラにジャンプし、復帰しない。
2663   curproc->state = ZOMBIE;
2664   sched();
2665   panic("zombie exit");
2666 }
2667
2668 // 子プロセスが終了してpidを返すのを待つ。
2669 // このプロセスが子プロセスを持たない場合は -1 を返す。
2670 int
2671 wait(void)
2672 {
2673   struct proc *p;
2674   int havekids, pid;
2675   struct proc *curproc = myproc();
2676
2677   acquire(&ptable.lock);
2678   for(;;){
2679     // テーブルを走査して終了した子プロセスを探す。
2680     havekids = 0;
2681     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2682       if(p->parent != curproc)
2683         continue;
2684       havekids = 1;
2685       if(p->state == ZOMBIE){
2686         // 見つけた。
2687         pid = p->pid;
2688         kfree(p->kstack);
2689         p->kstack = 0;
2690         freevm(p->pgdir);
2691         p->pid = 0;
2692         p->parent = 0;
2693         p->name[0] = 0;
2694         p->killed = 0;
2695         p->state = UNUSED;
2696         release(&ptable.lock);
2697         return pid;
2698       }
2699     }
```

```
2700     // 子プロセスを持っていなければ待っても意味がない。
2701     if(!havekids || curproc->killed){
2702       release(&ptable.lock);
2703       return -1;
2704     }
2705
2706     // 子プロセスの終了を待つ。(proc_exitのwakeup1コールを参照)
2707     sleep(curproc, &ptable.lock);
2708   }
2709 }
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
```



```
2750 // CPUごとのプロセススケジューラ。
2751 // 各CPUは自身を設定した後 scheduler() を呼び出す。
2752 // スケジューラは復帰しない。ループして以下を行う:
2753 //  - 実行するプロセスを選択する
2754 //  - swtchを呼び出してそのプロセスの実行を開始する
2755 //  - 最終的にそのプロセスはスswtchを呼び出して
2756 //      スケジューラに制御を戻す
2757 void
2758 scheduler(void)
2759 {
2760   struct proc *p;
2761   struct cpu *c = mycpu();
2762   c->proc = 0;
2763
2764   for(;;){
2765     // このプロセッサ上での割り込みを有効にする。
2766     sti();
2767
2768     // プロセステーブルを走査して実行するプロセスを探す、
2769     acquire(&ptable.lock);
2770     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2771       if(p->state != RUNNABLE)
2772         continue;
2773
2774       // 選択したプロセスにスイッチする。ptable.lockを解放して、
2775       // スケジューラに戻る前に再度ロックするのは
2776       // プロセスの仕事である。
2777       c->proc = p;
2778       switchuvm(p);
2779       p->state = RUNNING;
2780
2781       swtch(&(c->scheduler), p->context);
2782       switchkvm();
2783
2784       // ここではプロセスは実行を終えている。
2785       // プロセスはここに戻る前に自分でp->stateを変更しているするはずだ。
2786       c->proc = 0;
2787     }
2788     release(&ptable.lock);
2789
2790   }
2791 }
2792
2793
2794
2795
2796
2797
2798
2799
```

```
2800 // スケジューラに入る。ptable.lockだけを保持し、
2801 // proc->stateが変更されていなければならない。
2802 // intenaは、このカーネルスレッドの属性であり
2803 // このCPUの属性ではないので、intenaの保存と復元を行う。
2804 // 本来ならproc->intenaとproc->ncliとするべきだが、それだと
2805 // ロックは保持されているがプロセスがないような場所で
2806 // まれに破綻する可能性がある。
2807 void
2808 sched(void)
2809 {
2810   int intena;
2811   struct proc *p = myproc();
2812
2813   if(!holding(&ptable.lock))
2814     panic("sched ptable.lock");
2815   if(mycpu()->ncli != 1)
2816     panic("sched locks");
2817   if(p->state == RUNNING)
2818     panic("sched running");
2819   if(readeflags()&FL_IF)
2820     panic("sched interruptible");
2821   intena = mycpu()->intena;
2822   swtch(&p->context, mycpu()->scheduler);
2823   mycpu()->intena = intena;
2824 }
2825
2826 // 1回のスケジュール処理ごとにCPUを明け渡す。
2827 void
2828 yield(void)
2829 {
2830   acquire(&ptable.lock);
2831   myproc()->state = RUNNABLE;
2832   sched();
2833   release(&ptable.lock);
2834 }
2835
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
```



```
2850 // scheduler()でスケジュールされる一番最初のフォークの子プロセスは
2851 // ここにスイッチする。ユーザ空間に「復帰する」。
2852 void
2853 forkret(void)
2854 {
2855   static int first = 1;
2856   // スケジューラからの ptable.lock をまだ保持しているので
2857   release(&ptable.lock);
2858
2859   if (first) {
2860     // ある種の初期化関数は通常のプロセスのコンテキストで実行
2861     // されなければならない ( たとえば、スリープのコールなど ) 。
2862     // そのため、main()からは実行することができない。
2863     first = 0;
2864     iinit(ROOTDEV);
2865     initlog(ROOTDEV);
2866   }
2867
2868   // 「呼び出し元」に戻るが、実際はtrapretに戻る ( allocprocを参照 ) 。
2869 }
2870
2871 // アトミックにロックを解放し、chanでスリープする。
2872 // 起床した時にロックを再度獲得する。
2873 void
2874 sleep(void *chan, struct spinlock *lk)
2875 {
2876   struct proc *p = myproc();
2877
2878   if(p == 0)
2879     panic("sleep");
2880
2881   if(lk == 0)
2882     panic("sleep without lk");
2883
2884   // p->stateを変更し、schedを呼び出すために
2885   // ptable.lockを獲得しなければならない。
2886   // ptable.lockを保持すれば，起こし忘れが
2887   // ないことが保証される ( wakeupはptable.lockが
2888   // ロックされた状態で実行する ) ので，
2889   // lkを解放しても問題はない。
2890   if(lk != &ptable.lock){
2891     acquire(&ptable.lock);
2892     release(lk);
2893   }
2894   // スリープに入る。
2895   p->chan = chan;
2896   p->state = SLEEPING;
2897
2898   sched();
2899
```

```
2900   // 後片付けをする。
2901   p->chan = 0;
2902
2903   // 元々保持していたロックを再度獲得する。
2904   if(lk != &ptable.lock){
2905     release(&ptable.lock);
2906     acquire(lk);
2907   }
2908 }
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
```



```
2950 // chanでスリープしているすべてのプロセスを起床させる。
2951 // ptable.lockを保持していなければならない。
2952 static void
2953 wakeup1(void *chan)
2954 {
2955   struct proc *p;
2956
2957   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2958     if(p->state == SLEEPING && p->chan == chan)
2959       p->state = RUNNABLE;
2960 }
2961
2962 // chanでスリープしているすべてのプロセスを起床させる。
2963 void
2964 wakeup(void *chan)
2965 {
2966   acquire(&ptable.lock);
2967   wakeup1(chan);
2968   release(&ptable.lock);
2969 }
2970
2971 // 指定されたpidを持つプロセスをkillする。
2972 // プロセスはユーザ空間に復帰するまでは
2973 // 終了しない ( trap.cのtrapを参照 ) 。
2974 int
2975 kill(int pid)
2976 {
2977   struct proc *p;
2978
2979   acquire(&ptable.lock);
2980   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2981     if(p->pid == pid){
2982       p->killed = 1;
2983       // 必要であれば寝ているプロセスを起床させる。
2984       if(p->state == SLEEPING)
2985         p->state = RUNNABLE;
2986       release(&ptable.lock);
2987       return 0;
2988     }
2989   }
2990   release(&ptable.lock);
2991   return -1;
2992 }
2993
2994
2995
2996
2997
2998
2999
```

```
3000 // プロセス一覧をコンソールに出力する。デバッグ用。
3001 // ユーザがコンソールで^Pとタイプすると実行する。
3002 // スタックしたマシンをさらに割り込ませないようにロックはしない。
3003 void
3004 procdump(void)
3005 {
3006   static char *states[] = {
3007   [UNUSED]    "unused",
3008   [EMBRYO]    "embryo",
3009   [SLEEPING]  "sleep ",
3010   [RUNNABLE]  "runble",
3011   [RUNNING]   "run   ",
3012   [ZOMBIE]    "zombie"
3013   };
3014   int i;
3015   struct proc *p;
3016   char *state;
3017   uint pc[10];
3018
3019   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
3020     if(p->state == UNUSED)
3021       continue;
3022     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
3023       state = states[p->state];
3024     else
3025       state = "???";
3026     cprintf("%d %s %s", p->pid, state, p->name);
3027     if(p->state == SLEEPING){
3028       getcallerpcs((uint*)p->context->ebp+2, pc);
3029       for(i=0; i<10 && pc[i] != 0; i++)
3030         cprintf(" %p", pc[i]);
3031     }
3032     cprintf("\n");
3033   }
3034 }
3035
3036
3037
3038
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
```



```
3050 # コンテキストスイッチ
3051 #   void swtch(struct context **old, struct context *new);
3052 #
3053 # 現在のレジスタをスタックに保存し、struct contextを
3054 # 作成してそのアドレスを*oldに保存する。スタックをnewに
3055 # 切り替え、以前に保存していたレジスタをポップする。
3056
3057 .globl swtch
3058 swtch:
3059   movl 4(%esp), %eax       # old
3060   movl 8(%esp), %edx       # new
3061
3062   # 旧callee-saveレジスタを保存
3063   pushl %ebp
3064   pushl %ebx
3065   pushl %esi
3066   pushl %edi
3067
3068   # スタックを切り替える
3069   movl %esp, (%eax)
3070   movl %edx, %esp
3071
3072   # 新callee-saveレジスタをロード
3073   popl %edi
3074   popl %esi
3075   popl %ebx
3076   popl %ebp
3077   ret
3078
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
```

```
3100 // 物理メモリアロケータ。ユーザプロセス、カーネルスタック、
3101 // ページテーブルページ、パイプバッファのためのメモリの割り当てを行う。
3102 // 4096バイトのページを割り当てる
3103
3104 #include "types.h"
3105 #include "defs.h"
3106 #include "param.h"
3107 #include "memlayout.h"
3108 #include "mmu.h"
3109 #include "spinlock.h"
3110
3111 void freerange(void *vstart, void *vend);
3112 extern char end[]; // ELFファイルからロードされたカーネルに続く最初のアドレス。
3113                    // kernel.ldにあるカーネルリンカスクリプトで定義されている
3114
3115 struct run {
3116   struct run *next;
3117 };
3118
3119 struct {
3120   struct spinlock lock;
3121   int use_lock;
3122   struct run *freelist;
3123 } kmem;
3124
3125 // 初期化は２段階で行われる。
3126 // 1. main()はkinit1を呼び出す。この時にはまだentrypgdirが使用されており、
3127 // entrypgdirでマッピングされたページを空きリストに設定する。
3128 // 2. main()はkinit2()を呼び出す。すべてのコアで物理ページをマッピングする
3129 // 完全なページテーブルをインストールした後に、残りの物理ページを初期化する。
3130 void
3131 kinit1(void *vstart, void *vend)
3132 {
3133   initlock(&kmem.lock, "kmem");
3134   kmem.use_lock = 0;
3135   freerange(vstart, vend);
3136 }
3137
3138 void
3139 kinit2(void *vstart, void *vend)
3140 {
3141   freerange(vstart, vend);
3142   kmem.use_lock = 1;
3143 }
3144
3145
3146
3147
3148
3149
```



```
3150 void
3151 freerange(void *vstart, void *vend)
3152 {
3153   char *p;
3154   p = (char*)PGROUNDUP((uint)vstart);
3155   for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
3156     kfree(p);
3157 }
3158
3159 // vで指し示される物理メモリのページを解放する。
3160 // vは通常、kalloc()の呼び出しで返されたポインタである。
3161 // （ 例外は、アロケータを初期化する場合である。
3162 // 上のkinitを参照 ）
3163 void
3164 kfree(char *v)
3165 {
3166   struct run *r;
3167
3168   if((uint)v % PGSIZE ¦¦ v < end ¦¦ V2P(v) >= PHYSTOP)
3169     panic("kfree");
3170
3171   // ダングリング参照をキャッチするためにジャンクで満たす
3172   memset(v, 1, PGSIZE);
3173
3174   if(kmem.use_lock)
3175     acquire(&kmem.lock);
3176   r = (struct run*)v;
3177   r->next = kmem.freelist;
3178   kmem.freelist = r;
3179   if(kmem.use_lock)
3180     release(&kmem.lock);
3181 }
3182
3183 // 4096バイト単位の物理メモリページを1ページ割り当てる。
3184 // カーネルが利用可能なポインタを返す。
3185 // メモリを割り当てられなかった場合は、0を返す。
3186 char*
3187 kalloc(void)
3188 {
3189   struct run *r;
3190
3191   if(kmem.use_lock)
3192     acquire(&kmem.lock);
3193   r = kmem.freelist;
3194   if(r)
3195     kmem.freelist = r->next;
3196   if(kmem.use_lock)
3197     release(&kmem.lock);
3198   return (char*)r;
3199 }
```

```
3200 // x86のトラップと割り込み関係の定数
3201
3202 // プロセッサで定義:
3203 #define T_DIVIDE         0      // 除算エラー
3204 #define T_DEBUG          1      // デバッグ例外
3205 #define T_NMI            2      // NMI割り込み
3206 #define T_BRKPT          3      // ブレークポイント
3207 #define T_OFLOW          4      // オーバーフロー
3208 #define T_BOUND          5      // BOUND範囲超過
3209 #define T_ILLOP          6      // 無効オペコード
3210 #define T_DEVICE         7      // デバイス使用不可
3211 #define T_DBLFLT         8      // ダブルフォルト
3212 // #define T_COPROC      9      // 予約済（486以降未使用）
3213 #define T_TSS           10      // 無効TSS
3214 #define T_SEGNP         11      // セグメント不在
3215 #define T_STACK         12      // スタックフォルト例外
3216 #define T_GPFLT         13      // 一般保護例外
3217 #define T_PGFLT         14      // ページフォルト
3218 // #define T_RES        15      // 予約済
3219 #define T_FPERR         16      // 浮動小数点エラー
3220 #define T_ALIGN         17      // アライメントチェック
3221 #define T_MCHK          18      // マシンチェック
3222 #define T_SIMDERR       19      // SIMD浮動小数点エラー
3223
3224 // 以下の値は任意に選んだものであるが、プロセッサが定義している
3225 // 例外・割り込みベクタと重ならないように注意している。
3226 #define T_SYSCALL       64      // システムコール
3227 #define T_DEFAULT      500      // キャッチオール
3228
3229 #define T_IRQ0          32      // IRQ 0 はint T_IRQに相当する
3230
3231 #define IRQ_TIMER        0
3232 #define IRQ_KBD          1
3233 #define IRQ_COM1         4
3234 #define IRQ_IDE         14
3235 #define IRQ_ERROR       19
3236 #define IRQ_SPURIOUS    31
3237
3238
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
```



```
3250 #!/usr/bin/perl -w
3251
3252 # トラップと割り込みのエントリポイントであるvectors.Sを生成する。
3253 # 割り込み番号ごとに1つエントリポイントがなければならない。
3254 # なぜなら、それ以外にtrap()が割り込み番号を知る方法がない
3255 # からである。
3256
3257 print "# generated by vectors.pl - do not edit\n";
3258 print "# handlers\n";
3259 print ".globl alltraps\n";
3260 for(my $i = 0; $i < 256; $i++){
3261     print ".globl vector$i\n";
3262     print "vector$i:\n";
3263     if(!($i == 8 ¦¦ ($i >= 10 && $i <= 14) ¦¦ $i == 17)){
3264         print "  pushl \$0\n";
3265     }
3266     print "  pushl \$$i\n";
3267     print "  jmp alltraps\n";
3268 }
3269
3270 print "\n# vector table\n";
3271 print ".data\n";
3272 print ".globl vectors\n";
3273 print "vectors:\n";
3274 for(my $i = 0; $i < 256; $i++){
3275     print "  .long vector$i\n";
3276 }
3277
3278 # 出力例:
3279 #   # handlers
3280 #   .globl alltraps
3281 #   .globl vector0
3282 #   vector0:
3283 #     pushl $0
3284 #     pushl $0
3285 #     jmp alltraps
3286 #   ...
3287 #
3288 #   # vector table
3289 #   .data
3290 #   .globl vectors
3291 #   vectors:
3292 #     .long vector0
3293 #     .long vector1
3294 #     .long vector2
3295 #   ...
3296
3297
3298
3299
```

```
3300 #include "mmu.h"
3301
3302   # vectors.S はすべてのトラップをここに送る。
3303 .globl alltraps
3304 alltraps:
3305   # トラップフレームを構築する。
3306   pushl %ds
3307   pushl %es
3308   pushl %fs
3309   pushl %gs
3310   pushal
3311
3312   # データセグメントを設定する。
3313   movw $(SEG_KDATA<<3), %ax
3314   movw %ax, %ds
3315   movw %ax, %es
3316
3317   # trap(tf)を呼び出す。ここで、tf=%esp
3318   pushl %esp
3319   call trap
3320   addl $4, %esp
3321
3322   # 復帰はtrapretにフォールスルーする...
3323 .globl trapret
3324 trapret:
3325   popal
3326   popl %gs
3327   popl %fs
3328   popl %es
3329   popl %ds
3330   addl $0x8, %esp  # トラップ番号とエラーコード
3331   iret
3332
3333
3334
3335
3336
3337
3338
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349
```



```
3350 #include "types.h"
3351 #include "defs.h"
3352 #include "param.h"
3353 #include "memlayout.h"
3354 #include "mmu.h"
3355 #include "proc.h"
3356 #include "x86.h"
3357 #include "traps.h"
3358 #include "spinlock.h"
3359
3360 // 割り込みディスクリプタテーブル ( すべてのCPUで共有される )
3361 struct gatedesc idt[256];
3362 extern uint vectors[];  // vectors.Sで設定: 256エントリポインタの配列
3363 struct spinlock tickslock;
3364 uint ticks;
3365
3366 void
3367 tvinit(void)
3368 {
3369   int i;
3370
3371   for(i = 0; i < 256; i++)
3372     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3373   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3374
3375   initlock(&tickslock, "time");
3376 }
3377
3378 void
3379 idtinit(void)
3380 {
3381   lidt(idt, sizeof(idt));
3382 }
3383
3384
3385
3386
3387
3388
3389
3390
3391
3392
3393
3394
3395
3396
3397
3398
3399
```

```
3400 void
3401 trap(struct trapframe *tf)
3402 {
3403   if(tf->trapno == T_SYSCALL){
3404     if(myproc()->killed)
3405       exit();
3406     myproc()->tf = tf;
3407     syscall();
3408     if(myproc()->killed)
3409       exit();
3410     return;
3411   }
3412
3413   switch(tf->trapno){
3414   case T_IRQ0 + IRQ_TIMER:
3415     if(cpuid() == 0){
3416       acquire(&tickslock);
3417       ticks++;
3418       wakeup(&ticks);
3419       release(&tickslock);
3420     }
3421     lapiceoi();
3422     break;
3423   case T_IRQ0 + IRQ_IDE:
3424     ideintr();
3425     lapiceoi();
3426     break;
3427   case T_IRQ0 + IRQ_IDE+1:
3428     // BochsはスプリアスIDE1割り込みを生成する。
3429     break;
3430   case T_IRQ0 + IRQ_KBD:
3431     kbdintr();
3432     lapiceoi();
3433     break;
3434   case T_IRQ0 + IRQ_COM1:
3435     uartintr();
3436     lapiceoi();
3437     break;
3438   case T_IRQ0 + 7:
3439   case T_IRQ0 + IRQ_SPURIOUS:
3440     cprintf("cpu%d: spurious interrupt at %x:%x\n",
3441             cpuid(), tf->cs, tf->eip);
3442     lapiceoi();
3443     break;
3444
3445
3446
3447
3448
3449
```



```
3450   default:
3451     if(myproc() == 0 ¦¦ (tf->cs&3) == 0){
3452       // カーネルで発生。OSの問題に違いない。
3453       cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
3454               tf->trapno, cpuid(), tf->eip, rcr2());
3455       panic("trap");
3456     }
3457     // ユーザ空間で発生。プロセスが不正を行ったようだ。
3458     cprintf("pid %d %s: trap %d err %d on cpu %d "
3459             "eip 0x%x addr 0x%x--kill proc\n",
3460             myproc()->pid, myproc()->name, tf->trapno,
3461             tf->err, cpuid(), tf->eip, rcr2());
3462     myproc()->killed = 1;
3463   }
3464
3465   // プロセスがkillされており、ユーザ空間のプロセスであれば、
3466   // プロセスを終了させる ( カーネルで実行中の場合は、通常のシステム
3467   // コールの復帰を確認するまで実行を継続させる ) 。
3468   if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
3469     exit();
3470
3471   // クロック割り込みの場合はCPUを放棄させる。
3472   // (訳注: Obsolute)割り込みがロックの保持中にあったか否かをnlockでチェックする必要があるだ
3473   if(myproc() && myproc()->state == RUNNING &&
3474      tf->trapno == T_IRQ0+IRQ_TIMER)
3475     yield();
3476
3477   // yield中にプロセスがkillされたか否かをチェックする。
3478   if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
3479     exit();
3480 }
3481
3482
3483
3484
3485
3486
3487
3488
3489
3490
3491
3492
3493
3494
3495
3496
3497
3498
3499
```

```
3500 // システムコール番号
3501 #define SYS_fork    1
3502 #define SYS_exit    2
3503 #define SYS_wait    3
3504 #define SYS_pipe    4
3505 #define SYS_read    5
3506 #define SYS_kill    6
3507 #define SYS_exec    7
3508 #define SYS_fstat   8
3509 #define SYS_chdir   9
3510 #define SYS_dup    10
3511 #define SYS_getpid 11
3512 #define SYS_sbrk   12
3513 #define SYS_sleep  13
3514 #define SYS_uptime 14
3515 #define SYS_open   15
3516 #define SYS_write  16
3517 #define SYS_mknod  17
3518 #define SYS_unlink 18
3519 #define SYS_link   19
3520 #define SYS_mkdir  20
3521 #define SYS_close  21
3522
3523
3524
3525
3526
3527
3528
3529
3530
3531
3532
3533
3534
3535
3536
3537
3538
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
3549
```



```
3550 #include "types.h"
3551 #include "defs.h"
3552 #include "param.h"
3553 #include "memlayout.h"
3554 #include "mmu.h"
3555 #include "proc.h"
3556 #include "x86.h"
3557 #include "syscall.h"
3558
3559 // ユーザコードは INT T_SYSCALL でシステムコールを行う。
3560 // システムコール番号は %eax におく。
3561 // ユーザコールからCライブラリのシステムコール関数への引数はスタックに積まれる。
3562 // 保存されるユーザ%espは、保存されるプログラムカウンタを指しており、
3563 // 次が最初の引数である。
3564
3565 // カレントプロセスからaddrにあるintを取り出す。
3566 int
3567 fetchint(uint addr, int *ip)
3568 {
3569   struct proc *curproc = myproc();
3570
3571   if(addr >= curproc->sz ¦¦ addr+4 > curproc->sz)
3572     return -1;
3573   *ip = *(int*)(addr);
3574   return 0;
3575 }
3576
3577 // カレントプロセスからaddrにあるヌル終端文字列を取り出す。
3578 // 実際には文字列はコピーしない。*ppがそれを指しｦすように設定するだけである。
3579 // 文字列の長さ ( ヌルを含まない ) を返す。
3580 int
3581 fetchstr(uint addr, char **pp)
3582 {
3583   char *s, *ep;
3584   struct proc *curproc = myproc();
3585
3586   if(addr >= curproc->sz)
3587     return -1;
3588   *pp = (char*)addr;
3589   ep = (char*)curproc->sz;
3590   for(s = *pp; s < ep; s++){
3591     if(*s == 0)
3592       return s - *pp;
3593   }
3594   return -1;
3595 }
3596
3597
3598
3599
```

```
3600 // n番目の32ビットシステムコール引数を取り出す。
3601 int
3602 argint(int n, int *ip)
3603 {
3604   return fetchint((myproc()->tf->esp) + 4 + 4*n, ip);
3605 }
3606
3607 // n番目のワードサイズ(32bit)のシステムコール引数をsizeバイトの
3608 // メモリブロックを指しぇすポインタとして取り出す。
3609 // ポインタがプロセスのアドレス空間にあるかチェックする。
3610 int
3611 argptr(int n, char **pp, int size)
3612 {
3613   int i;
3614   struct proc *curproc = myproc();
3615
3616   if(argint(n, &i) < 0)
3617     return -1;
3618   if(size < 0 ¦¦ (uint)i >= curproc->sz ¦¦ (uint)i+size > curproc->sz)
3619     return -1;
3620   *pp = (char*)i;
3621   return 0;
3622 }
3623
3624 // n番目のワードサイズのシステムコール引数を文字列ポインタとして取り出す。
3625 // ポインタが有効かつ文字列がヌル終端であることをチェックする。
3626 // （共有の書き込み可能メモリがないので、このチェックのために
3627 // カーネルが使用する文字列を変更できない）
3628 int
3629 argstr(int n, char **pp)
3630 {
3631   int addr;
3632   if(argint(n, &addr) < 0)
3633     return -1;
3634   return fetchstr(addr, pp);
3635 }
3636
3637
3638
3639
3640
3641
3642
3643
3644
3645
3646
3647
3648
3649
```



```
3650 extern int sys_chdir(void);
3651 extern int sys_close(void);
3652 extern int sys_dup(void);
3653 extern int sys_exec(void);
3654 extern int sys_exit(void);
3655 extern int sys_fork(void);
3656 extern int sys_fstat(void);
3657 extern int sys_getpid(void);
3658 extern int sys_kill(void);
3659 extern int sys_link(void);
3660 extern int sys_mkdir(void);
3661 extern int sys_mknod(void);
3662 extern int sys_open(void);
3663 extern int sys_pipe(void);
3664 extern int sys_read(void);
3665 extern int sys_sbrk(void);
3666 extern int sys_sleep(void);
3667 extern int sys_unlink(void);
3668 extern int sys_wait(void);
3669 extern int sys_write(void);
3670 extern int sys_uptime(void);
3671
3672 static int (*syscalls[])(void) = {
3673 [SYS_fork]    sys_fork,
3674 [SYS_exit]    sys_exit,
3675 [SYS_wait]    sys_wait,
3676 [SYS_pipe]    sys_pipe,
3677 [SYS_read]    sys_read,
3678 [SYS_kill]    sys_kill,
3679 [SYS_exec]    sys_exec,
3680 [SYS_fstat]   sys_fstat,
3681 [SYS_chdir]   sys_chdir,
3682 [SYS_dup]     sys_dup,
3683 [SYS_getpid]  sys_getpid,
3684 [SYS_sbrk]    sys_sbrk,
3685 [SYS_sleep]   sys_sleep,
3686 [SYS_uptime]  sys_uptime,
3687 [SYS_open]    sys_open,
3688 [SYS_write]   sys_write,
3689 [SYS_mknod]   sys_mknod,
3690 [SYS_unlink]  sys_unlink,
3691 [SYS_link]    sys_link,
3692 [SYS_mkdir]   sys_mkdir,
3693 [SYS_close]   sys_close,
3694 };
3695
3696
3697
3698
3699
```

```
3700 void
3701 syscall(void)
3702 {
3703   int num;
3704   struct proc *curproc = myproc();
3705
3706   num = curproc->tf->eax;
3707   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
3708     curproc->tf->eax = syscalls[num]();
3709   } else {
3710     cprintf("%d %s: unknown sys call %d\n",
3711             curproc->pid, curproc->name, num);
3712     curproc->tf->eax = -1;
3713   }
3714 }
3715
3716
3717
3718
3719
3720
3721
3722
3723
3724
3725
3726
3727
3728
3729
3730
3731
3732
3733
3734
3735
3736
3737
3738
3739
3740
3741
3742
3743
3744
3745
3746
3747
3748
3749
```



```
3750 #include "types.h"
3751 #include "x86.h"
3752 #include "defs.h"
3753 #include "date.h"
3754 #include "param.h"
3755 #include "memlayout.h"
3756 #include "mmu.h"
3757 #include "proc.h"
3758
3759 int
3760 sys_fork(void)
3761 {
3762   return fork();
3763 }
3764
3765 int
3766 sys_exit(void)
3767 {
3768   exit();
3769   return 0;  // ここには来ない
3770 }
3771
3772 int
3773 sys_wait(void)
3774 {
3775   return wait();
3776 }
3777
3778 int
3779 sys_kill(void)
3780 {
3781   int pid;
3782
3783   if(argint(0, &pid) < 0)
3784     return -1;
3785   return kill(pid);
3786 }
3787
3788 int
3789 sys_getpid(void)
3790 {
3791   return myproc()->pid;
3792 }
3793
3794
3795
3796
3797
3798
3799
```

```
3800 int
3801 sys_sbrk(void)
3802 {
3803   int addr;
3804   int n;
3805
3806   if(argint(0, &n) < 0)
3807     return -1;
3808   addr = myproc()->sz;
3809   if(growproc(n) < 0)
3810     return -1;
3811   return addr;
3812 }
3813
3814 int
3815 sys_sleep(void)
3816 {
3817   int n;
3818   uint ticks0;
3819
3820   if(argint(0, &n) < 0)
3821     return -1;
3822   acquire(&tickslock);
3823   ticks0 = ticks;
3824   while(ticks - ticks0 < n){
3825     if(myproc()->killed){
3826       release(&tickslock);
3827       return -1;
3828     }
3829     sleep(&ticks, &tickslock);
3830   }
3831   release(&tickslock);
3832   return 0;
3833 }
3834
3835 // スタート以来、クロックティック割り込みが
3836 // 何回生じたかを返す。
3837 int
3838 sys_uptime(void)
3839 {
3840   uint xticks;
3841
3842   acquire(&tickslock);
3843   xticks = ticks;
3844   release(&tickslock);
3845   return xticks;
3846 }
3847
3848
3849
```



```
3850 struct buf {
3851   int flags;
3852   uint dev;
3853   uint blockno;
3854   struct sleeplock lock;
3855   uint refcnt;
3856   struct buf *prev; // LRUキャッシュリスト
3857   struct buf *next;
3858   struct buf *qnext; // ディスクキュー
3859   uchar data[BSIZE];
3860 };
3861 #define B_VALID 0x2  // バッファはディスクから読み込まれている
3862 #define B_DIRTY 0x4  // バッファをディスクに書き込む必要がある
3863
3864
3865
3866
3867
3868
3869
3870
3871
3872
3873
3874
3875
3876
3877
3878
3879
3880
3881
3882
3883
3884
3885
3886
3887
3888
3889
3890
3891
3892
3893
3894
3895
3896
3897
3898
3899
```

```
3900 // プロセス用の長期ロック
3901 struct sleeplock {
3902   uint locked;       // このロックは保持されているか?
3903   struct spinlock lk; // このスリープロックを保護するスピンロック
3904
3905   // デバッグ用:
3906   char *name;        // ロックの名前
3907   int pid;           // ロックを保持しているプロセス
3908 };
3909
3910
3911
3912
3913
3914
3915
3916
3917
3918
3919
3920
3921
3922
3923
3924
3925
3926
3927
3928
3929
3930
3931
3932
3933
3934
3935
3936
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
```



```
3950 #define O_RDONLY  0x000
3951 #define O_WRONLY  0x001
3952 #define O_RDWR    0x002
3953 #define O_CREATE  0x200
3954
3955
3956
3957
3958
3959
3960
3961
3962
3963
3964
3965
3966
3967
3968
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3979
3980
3981
3982
3983
3984
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999
```

```
4000 #define T_DIR  1   // ディレクトリ
4001 #define T_FILE 2   // ファイル
4002 #define T_DEV  3   // デバイス
4003
4004 struct stat {
4005   short type;  // ファイルの種類
4006   int dev;     // ファイルシステムのディスク装置
4007   uint ino;    // inode番号
4008   short nlink; // ファイルへのリンク数
4009   uint size;   // ファイルのサイズ ( 単位はバイト )
4010 };
4011
4012
4013
4014
4015
4016
4017
4018
4019
4020
4021
4022
4023
4024
4025
4026
4027
4028
4029
4030
4031
4032
4033
4034
4035
4036
4037
4038
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4049
```



```
4050 // ディスク上のファイルシステムフォーマット。
4051 // カーネルとユーザプログラムは共にこのヘッダファイルを使用する。
4052
4053
4054 #define ROOTINO 1  // ルートinode番号
4055 #define BSIZE 512  // ブロックサイズ
4056
4057 // ディスクレイアウト:
4058 // [ ブートブロック ¦ スーパーブロック ¦ ログ ¦ inodeブロック ¦
4059 //                              空きビットマップ ¦ データブロック ]
4060 //
4061 // mkfs はスーパーブロックを計算し、初期ファイルシステムを構築する。
4062 // スーパーブロックはディスクレイアウトを記述する:
4063 struct superblock {
4064   uint size;         // ファイルシステムイメージのサイズ ( 単位はブロック )
4065   uint nblocks;      // データブロックの数
4066   uint ninodes;      // inodeの数
4067   uint nlog;         // ログブロックの数
4068   uint logstart;     // 先頭のログブロックのブロック番号
4069   uint inodestart;   // 先頭のinodeブロックのブロック番号
4070   uint bmapstart;    // 先頭の空きマップブロックのブロック番号
4071 };
4072
4073 #define NDIRECT 12
4074 #define NINDIRECT (BSIZE / sizeof(uint))
4075 #define MAXFILE (NDIRECT + NINDIRECT)
4076
4077 // オンディスク inode 構造体
4078 struct dinode {
4079   short type;           // ファイルの種類
4080   short major;          // メジャーデバイス番号 ( T_DEV のみ)
4081   short minor;          // マイナーデバイス番号 (T_DEV のみ)
4082   short nlink;          // ファイルシステム内のinodeへのリンクの数
4083   uint size;            // ファイルのサイズ ( 単位はバイト )
4084   uint addrs[NDIRECT+1];   // データブロックアドレス
4085 };
4086
4087
4088
4089
4090
4091
4092
4093
4094
4095
4096
4097
4098
4099
```

```
4100 // ブロックあたりのinode数
4101 #define IPB           (BSIZE / sizeof(struct dinode))
4102
4103 // inode iを含むブロック
4104 #define IBLOCK(i, sb)     ((i) / IPB + sb.inodestart)
4105
4106 // ブロックあたりのBitmapビット数
4107 #define BPB           (BSIZE*8)
4108
4109 // ビットbを含んでいる空きマップのブロック
4110 #define BBLOCK(b, sb) (b/BPB + sb.bmapstart)
4111
4112 // ディレクトリは一連のdirent構造体を含むファイルである
4113 #define DIRSIZ 14
4114
4115 struct dirent {
4116   ushort inum;
4117   char name[DIRSIZ];
4118 };
4119
4120
4121
4122
4123
4124
4125
4126
4127
4128
4129
4130
4131
4132
4133
4134
4135
4136
4137
4138
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
```



```
4150 struct file {
4151   enum { FD_NONE, FD_PIPE, FD_INODE } type;
4152   int ref; // 参照カウント
4153   char readable;
4154   char writable;
4155   struct pipe *pipe;
4156   struct inode *ip;
4157   uint off;
4158 };
4159
4160
4161 // inodeのインメモリコピー
4162 struct inode {
4163   uint dev;           // デバイス番号
4164   uint inum;          // inode番号
4165   int ref;            // 参照カウント
4166   struct sleeplock lock; // 以下のすべてを保護する
4167   int valid;          // inodeはディスクから読み込まれているか?
4168
4169   short type;         // ディスクinodeのコピー
4170   short major;
4171   short minor;
4172   short nlink;
4173   uint size;
4174   uint addrs[NDIRECT+1];
4175 };
4176
4177 // メジャーデバイス番号をデバイス関数に
4178 // マッピングするテーブル
4179 struct devsw {
4180   int (*read)(struct inode*, char*, int);
4181   int (*write)(struct inode*, char*, int);
4182 };
4183
4184 extern struct devsw devsw[];
4185
4186 #define CONSOLE 1
4187
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199
```

```
4200 // 簡単なPIOベース（DMAを使用しない）IDEドライバのコード。
4201
4202 #include "types.h"
4203 #include "defs.h"
4204 #include "param.h"
4205 #include "memlayout.h"
4206 #include "mmu.h"
4207 #include "proc.h"
4208 #include "x86.h"
4209 #include "traps.h"
4210 #include "spinlock.h"
4211 #include "sleeplock.h"
4212 #include "fs.h"
4213 #include "buf.h"
4214
4215 #define SECTOR_SIZE   512
4216 #define IDE_BSY       0x80
4217 #define IDE_DRDY      0x40
4218 #define IDE_DF        0x20
4219 #define IDE_ERR       0x01
4220
4221 #define IDE_CMD_READ  0x20
4222 #define IDE_CMD_WRITE 0x30
4223 #define IDE_CMD_RDMUL 0xc4
4224 #define IDE_CMD_WRMUL 0xc5
4225
4226 // idequeue は現在ディスクを読み書きしているバッファを指している。
4227 // idequeue->qnext は次に処理するバッファを指している。
4228 // キューの操作中はidelockを保持しなければならない。
4229
4230 static struct spinlock idelock;
4231 static struct buf *idequeue;
4232
4233 static int havedisk1;
4234 static void idestart(struct buf*);
4235
4236 // IDEディスクの準備完了を待つ。
4237 static int
4238 idewait(int checkerr)
4239 {
4240   int r;
4241
4242   while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
4243     ;
4244   if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
4245     return -1;
4246   return 0;
4247 }
4248
4249
```



```
4250 void
4251 ideinit(void)
4252 {
4253   int i;
4254
4255   initlock(&idelock, "ide");
4256   ioapicenable(IRQ_IDE, ncpu - 1);
4257   idewait(0);
4258
4259   // ディスク1が存在するかチェックする。
4260   outb(0x1f6, 0xe0 | (1<<4));
4261   for(i=0; i<1000; i++){
4262     if(inb(0x1f7) != 0){
4263       havedisk1 = 1;
4264       break;
4265     }
4266   }
4267
4268   // ディスク0に戻す。
4269   outb(0x1f6, 0xe0 | (0<<4));
4270 }
4271
4272 // バッファに対するリクエストを開始する。呼び出し側はidelockの保持が必要。
4273 static void
4274 idestart(struct buf *b)
4275 {
4276   if(b == 0)
4277     panic("idestart");
4278   if(b->blockno >= FSSIZE)
4279     panic("incorrect blockno");
4280   int sector_per_block =  BSIZE/SECTOR_SIZE;
4281   int sector = b->blockno * sector_per_block;
4282   int read_cmd = (sector_per_block == 1) ? IDE_CMD_READ :  IDE_CMD_RDMUL;
4283   int write_cmd = (sector_per_block == 1) ? IDE_CMD_WRITE : IDE_CMD_WRMUL;
4284
4285   if (sector_per_block > 7) panic("idestart");
4286
4287   idewait(0);
4288   outb(0x3f6, 0);  // 割り込みを生成する
4289   outb(0x1f2, sector_per_block);  // セクターの数
4290   outb(0x1f3, sector & 0xff);
4291   outb(0x1f4, (sector >> 8) & 0xff);
4292   outb(0x1f5, (sector >> 16) & 0xff);
4293   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
4294   if(b->flags & B_DIRTY){
4295     outb(0x1f7, write_cmd);
4296     outsl(0x1f0, b->data, BSIZE/4);
4297   } else {
4298     outb(0x1f7, read_cmd);
4299   }
```

```
4300 }
4301
4302 // 割り込みハンドラ
4303 void
4304 ideintr(void)
4305 {
4306   struct buf *b;
4307
4308   // キューの先頭のバッファが処理対象のリクエストである。
4309   acquire(&idelock);
4310
4311   if((b = idequeue) == 0){
4312     release(&idelock);
4313     return;
4314   }
4315   idequeue = b->qnext;
4316
4317   // 必要であればデータを読み込む。
4318   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
4319     insl(0x1f0, b->data, BSIZE/4);
4320
4321   // このバッファを待機しているプロセスを起床させる。
4322   b->flags |= B_VALID;
4323   b->flags &= ~B_DIRTY;
4324   wakeup(b);
4325
4326   // キューの次のバッファのディスク処理を開始する。
4327   if(idequeue != 0)
4328     idestart(idequeue);
4329
4330   release(&idelock);
4331 }
4332
4333
4334
4335
4336
4337
4338
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
```



```
4350 // バッファとディスクを同期する。B_DIRTYがセットされていたら、バッファをディスクに書き込み、
4351 // B_DIRTYをクリアして、B_VALIDをセットする。B_DIRTYもB_VALIDもセットされていない場合は、
4352 // ディスクからバッファに読み込み、B_VALIDをセットする。
4353 void
4354 iderw(struct buf *b)
4355 {
4356   struct buf **pp;
4357
4358   if(!holdingsleep(&b->lock))
4359     panic("iderw: buf not locked");
4360   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
4361     panic("iderw: nothing to do");
4362   if(b->dev != 0 && !havedisk1)
4363     panic("iderw: ide disk 1 not present");
4364
4365   acquire(&idelock);
4366
4367   // bをidequeueに追加する。
4368   b->qnext = 0;
4369   for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
4370     ;
4371   *pp = b;
4372
4373   // 必要であればディスク処理を開始する。
4374   if(idequeue == b)
4375     idestart(b);
4376
4377   // リクエストが完了するのを待つ。
4378   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
4379     sleep(b, &idelock);
4380   }
4381
4382
4383   release(&idelock);
4384 }
4385
4386
4387
4388
4389
4390
4391
4392
4393
4394
4395
4396
4397
4398
4399
```

```
4400 // バッファキャッシュ。
4401 //
4402 // バッファキャッシュは、ディスクブロックコンテンツのキャッシュコピーを
4403 // 保持するバッファ構造体の連結リストである。ディスクブロックをメモリに
4404 // キャッシュすることで、ディスクのリード回数を減らす共に、
4405 // 複数のプロセスに使用されるディスクブロックに同期点を与える。
4406 //
4407 // インターフェース:
4408 // * 特定のディスクブロックを取得するには、breadを呼び出す。
4409 // * バッファデータをキャッシュした後、データをディスクに書き込みにはbwriteを呼び出す。
4410 // * バッファを使用する作業が終わったら、brelseを呼び出す。
4411 // * brelseの呼び出し後には、バッファを使用しない。
4412 // * バッファを使用できるのは一度に一つのプロセスだけである。
4413 //     そのため、必要以上にバッファを保持しないこと。
4414 //
4415 // この実装では内部で次の2つの状態フラグを使用する。
4416 // * B_VALID: バッファデータはディスクから読み込まれている。
4417 // * B_DIRTY: バッファデータは変更されており、
4418 //     ディスクに書き込む必要がある。
4419
4420 #include "types.h"
4421 #include "defs.h"
4422 #include "param.h"
4423 #include "spinlock.h"
4424 #include "sleeplock.h"
4425 #include "fs.h"
4426 #include "buf.h"
4427
4428 struct {
4429   struct spinlock lock;
4430   struct buf buf[NBUF];
4431
4432   // prev/nextでたどることができる、すべてのバッファからなる連結リスト。
4433   // head.nextはもっとも最近使用されたバッファである。
4434   struct buf head;
4435 } bcache;
4436
4437 void
4438 binit(void)
4439 {
4440   struct buf *b;
4441
4442   initlock(&bcache.lock, "bcache");
4443
4444
4445
4446
4447
4448
4449
```



```
4450   // バッファの連結リストを作成する。
4451   bcache.head.prev = &bcache.head;
4452   bcache.head.next = &bcache.head;
4453   for(b = bcache.buf; b < bcache.buf+NBUF; b++){
4454     b->next = bcache.head.next;
4455     b->prev = &bcache.head;
4456     initsleeplock(&b->lock, "buffer");
4457     bcache.head.next->prev = b;
4458     bcache.head.next = b;
4459   }
4460 }
4461
4462 // バッファキャッシュを走査して、指定したデバイスdevのブロックを探し出す。
4463 // 見つからなかった場合は、バッファを割り当てる。
4464 // いずれの場合も、ロックしたバッファを返す。
4465 static struct buf*
4466 bget(uint dev, uint blockno)
4467 {
4468   struct buf *b;
4469
4470   acquire(&bcache.lock);
4471
4472   // バッファはキャシュ済みか?
4473   for(b = bcache.head.next; b != &bcache.head; b = b->next){
4474     if(b->dev == dev && b->blockno == blockno){
4475       b->refcnt++;
4476       release(&bcache.lock);
4477       acquiresleep(&b->lock);
4478       return b;
4479     }
4480   }
4481
4482   // キャッシュされていない。未使用のバッファをリサイクルする。
4483   // refcnt==0であっても、B_DIRTYがセットされていたら、バッファは使用中である。
4484   // log.cでバッファが変更されたが、まだコミットされていないからである。
4485   for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
4486     if(b->refcnt == 0 && (b->flags & B_DIRTY) == 0) {
4487       b->dev = dev;
4488       b->blockno = blockno;
4489       b->flags = 0;
4490       b->refcnt = 1;
4491       release(&bcache.lock);
4492       acquiresleep(&b->lock);
4493       return b;
4494     }
4495   }
4496   panic("bget: no buffers");
4497 }
4498
4499
```

```
// 指定したブロックのコンテンツを含むバッファをロックして返す。
struct buf*
bread(uint dev, uint blockno)
{
  struct buf *b;

  b = bget(dev, blockno);
  if((b->flags & B_VALID) == 0) {
    iderw(b);
  }
  return b;
}

// バッファのコンテンツをディスクに書き込む。バッファはロックされていなければならない。
void
bwrite(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("bwrite");
  b->flags |= B_DIRTY;
  iderw(b);
}

// ロックされているバッファを解放する。
// MRUリストの先頭に移動させる。
void
brelse(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("brelse");

  releasesleep(&b->lock);

  acquire(&bcache.lock);
  b->refcnt--;
  if (b->refcnt == 0) {
    // このバッファを待機しているプロセスはない。
    b->next->prev = b->prev;
    b->prev->next = b->next;
    b->next = bcache.head.next;
    b->prev = &bcache.head;
    bcache.head.next->prev = b;
    bcache.head.next = b;
  }

  release(&bcache.lock);
}
```



```
// Blank page.
```

```
4600 // スリープロック。
4601
4602 #include "types.h"
4603 #include "defs.h"
4604 #include "param.h"
4605 #include "x86.h"
4606 #include "memlayout.h"
4607 #include "mmu.h"
4608 #include "proc.h"
4609 #include "spinlock.h"
4610 #include "sleeplock.h"
4611
4612 void
4613 initsleeplock(struct sleeplock *lk, char *name)
4614 {
4615   initlock(&lk->lk, "sleep lock");
4616   lk->name = name;
4617   lk->locked = 0;
4618   lk->pid = 0;
4619 }
4620
4621 void
4622 acquiresleep(struct sleeplock *lk)
4623 {
4624   acquire(&lk->lk);
4625   while (lk->locked) {
4626     sleep(lk, &lk->lk);
4627   }
4628   lk->locked = 1;
4629   lk->pid = myproc()->pid;
4630   release(&lk->lk);
4631 }
4632
4633 void
4634 releasesleep(struct sleeplock *lk)
4635 {
4636   acquire(&lk->lk);
4637   lk->locked = 0;
4638   lk->pid = 0;
4639   wakeup(lk);
4640   release(&lk->lk);
4641 }
4642
4643
4644
4645
4646
4647
4648
4649
```



```
4650 int
4651 holdingsleep(struct sleeplock *lk)
4652 {
4653   int r;
4654
4655   acquire(&lk->lk);
4656   r = lk->locked;
4657   release(&lk->lk);
4658   return r;
4659 }
4660
4661
4662
4663
4664
4665
4666
4667
4668
4669
4670
4671
4672
4673
4674
4675
4676
4677
4678
4679
4680
4681
4682
4683
4684
4685
4686
4687
4688
4689
4690
4691
4692
4693
4694
4695
4696
4697
4698
4699
```

```
4700 #include "types.h"
4701 #include "defs.h"
4702 #include "param.h"
4703 #include "spinlock.h"
4704 #include "sleeplock.h"
4705 #include "fs.h"
4706 #include "buf.h"
4707
4708 // 並列FSシステムコールを可能にするシンプルなロギングシステム。
4709 //
4710 // ログトランザクションには複数のFSシステムコールの更新内容が含まれている。
4711 // ロギングシステムはアクティブなFSシステムコールが存在しない場合にのみ
4712 // コミットを行う。したがって、コミットによりコミットされていない
4713 // システムコールの更新がディスクに書き込まれないかと心配する必要は
4714 // まったくない。
4715 //
4716 // システムコールはその開始と終了を知らせるためにbegin_op()/end_op()を
4717 // 呼び出さなければならない。通常、begin_op()は実行中のFSシステムコールの
4718 // カウントをインクリメントしただけで復帰する。
4719 // ただし、ログの枯渇が近いと判断した場合、最後の未処理のend_op()が
4720 // コミットされるまでbegin_op()はスリープする。
4721 //
4722 // ログはディスクブロックを含んでいる物理的なre-doログである。
4723 // オンディスクログフォーマットは次の通り:
4724 //   ヘッダブロック。ブロックA, B, C, ...のブロック番号を含んでいる
4725 //   ブロックA
4726 //   ブロックB
4727 //   ブロックC
4728 //   ...
4729 // ログの追加は同期的に行われる
4730
4731 // ヘッダブロックの内容。オンディスクヘッダブロックと
4732 // コミット前のログ出力ブロック番号をメモリ上で追跡するために使用される
4733 struct logheader {
4734   int n;
4735   int block[LOGSIZE];
4736 };
4737
4738 struct log {
4739   struct spinlock lock;
4740   int start;
4741   int size;
4742   int outstanding; // 実行中のFSシステムコールの数
4743   int committing;  // commit()中、待て
4744   int dev;
4745   struct logheader lh;
4746 };
4747
4748
4749
```



```
4750 struct log log;
4751
4752 static void recover_from_log(void);
4753 static void commit();
4754
4755 void
4756 initlog(int dev)
4757 {
4758   if (sizeof(struct logheader) >= BSIZE)
4759     panic("initlog: too big logheader");
4760
4761   struct superblock sb;
4762   initlock(&log.lock, "log");
4763   readsb(dev, &sb);
4764   log.start = sb.logstart;
4765   log.size = sb.nlog;
4766   log.dev = dev;
4767   recover_from_log();
4768 }
4769
4770 // コミットしたブロックをログからその本来の場所にコピーする。
4771 static void
4772 install_trans(void)
4773 {
4774   int tail;
4775
4776   for (tail = 0; tail < log.lh.n; tail++) {
4777     struct buf *lbuf = bread(log.dev, log.start+tail+1); // ログブロックを読み込む
4778     struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // dstを読み込む
4779     memmove(dbuf->data, lbuf->data, BSIZE);  // ブロックをdstにコピーする
4780     bwrite(dbuf);  // dstからディスクに書き込む
4781     brelse(lbuf);
4782     brelse(dbuf);
4783   }
4784 }
4785
4786 // ログヘッダをディスクからインメモリログヘッダに読み込む
4787 static void
4788 read_head(void)
4789 {
4790   struct buf *buf = bread(log.dev, log.start);
4791   struct logheader *lh = (struct logheader *) (buf->data);
4792   int i;
4793   log.lh.n = lh->n;
4794   for (i = 0; i < log.lh.n; i++) {
4795     log.lh.block[i] = lh->block[i];
4796   }
4797   brelse(buf);
4798 }
4799
```

```
4800 // インメモリログヘッダをディスクに書き込む。
4801 // ここで本当にカレントトランザクションが
4802 // コミットされることになる。
4803 static void
4804 write_head(void)
4805 {
4806   struct buf *buf = bread(log.dev, log.start);
4807   struct logheader *hb = (struct logheader *) (buf->data);
4808   int i;
4809   hb->n = log.lh.n;
4810   for (i = 0; i < log.lh.n; i++) {
4811     hb->block[i] = log.lh.block[i];
4812   }
4813   bwrite(buf);
4814   brelse(buf);
4815 }
4816
4817 static void
4818 recover_from_log(void)
4819 {
4820   read_head();
4821   install_trans(); // コミットされていたら、ログからディスクにコピーする
4822   log.lh.n = 0;
4823   write_head(); // ログをクリア
4824 }
4825
4826 // 各FSシステムコールの最初に呼び出される。
4827 void
4828 begin_op(void)
4829 {
4830   acquire(&log.lock);
4831   while(1){
4832     if(log.committing){
4833       sleep(&log, &log.lock);
4834     } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
4835       // この操作によりログスペースが枯渇するおそれがあるので、コミットされるまで待機する。
4836       sleep(&log, &log.lock);
4837     } else {
4838       log.outstanding += 1;
4839       release(&log.lock);
4840       break;
4841     }
4842   }
4843 }
4844
4845
4846
4847
4848
4849
```



```
4850 // 各FSシステムコールの最後に呼び出される。
4851 // これが最後の未処理の操作だった場合はコミットする。
4852 void
4853 end_op(void)
4854 {
4855   int do_commit = 0;
4856
4857   acquire(&log.lock);
4858   log.outstanding -= 1;
4859   if(log.committing)
4860     panic("log.committing");
4861   if(log.outstanding == 0){
4862     do_commit = 1;
4863     log.committing = 1;
4864   } else {
4865     // begin_op()がログスペースが空くのを待っている可能性がある。
4866     // log.outstandingをデクリメントしたことで予約スペースが
4867     // 減ったかもしれない。
4868     wakeup(&log);
4869   }
4870   release(&log.lock);
4871
4872   if(do_commit){
4873     // ロックを保持したままスリープすることは許されないので、
4874     // ロックを獲得せずにcommitを呼び出す。
4875     commit();
4876     acquire(&log.lock);
4877     log.committing = 0;
4878     wakeup(&log);
4879     release(&log.lock);
4880   }
4881 }
4882
4883 // 変更されたブロックをキャッシュからログにコピーする。
4884 static void
4885 write_log(void)
4886 {
4887   int tail;
4888
4889   for (tail = 0; tail < log.lh.n; tail++) {
4890     struct buf *to = bread(log.dev, log.start+tail+1); // ログブロック
4891     struct buf *from = bread(log.dev, log.lh.block[tail]); // キャッシュブロック
4892     memmove(to->data, from->data, BSIZE);
4893     bwrite(to);  // ログを書き込む
4894     brelse(from);
4895     brelse(to);
4896   }
4897 }
4898
4899
```

```
4900 static void
4901 commit()
4902 {
4903   if (log.lh.n > 0) {
4904     write_log();     // 変更されたブロックをキャッシュからログに書き込む
4905     write_head();    // ヘッダをディスクに書き込む -- 本当のコミット
4906     install_trans(); // 書き込み内容を本来の場所にコピーする
4907     log.lh.n = 0;
4908     write_head();    // トランザクションをログから消去する
4909   }
4910 }
4911
4912 // 呼び出し側でb->dataを変更した。それは、バッファ上で行われている。
4913 // ブロック番号を記録して、キャッシュのB_DIRTYフラグをたてる。
4914 // いずれ、commit()/write_log()がディスクへの書き込みを行う。
4915 //
4916 // log_write()はbwrite()の代わりに使用する。通常の使用法は次の通り:
4917 //   bp = bread(...)
4918 //   modify bp->data[]
4919 //   log_write(bp)
4920 //   brelse(bp)
4921 void
4922 log_write(struct buf *b)
4923 {
4924   int i;
4925
4926   if (log.lh.n >= LOGSIZE ¦¦ log.lh.n >= log.size - 1)
4927     panic("too big a transaction");
4928   if (log.outstanding < 1)
4929     panic("log_write outside of trans");
4930
4931   acquire(&log.lock);
4932   for (i = 0; i < log.lh.n; i++) {
4933     if (log.lh.block[i] == b->blockno)   // ログの統合
4934       break;
4935   }
4936   log.lh.block[i] = b->blockno;
4937   if (i == log.lh.n)
4938     log.lh.n++;
4939   b->flags ¦= B_DIRTY; // スワップアウトを防ぐ
4940   release(&log.lock);
4941 }
4942
4943
4944
4945
4946
4947
4948
4949
```



```
4950 // ファイルシステムの実装． 5階層:
4951 //   + ブロック: 生ディスクブロックのアロケータ
4952 //   + ログ: 多段階更新のクラッシュリカバリ
4953 //   + ファイル: inodeアロケータ、読み取り、書き込み、メタデータ
4954 //   + ディレクトリ: 特別なコンテンツ( inodeのリスト )を持つinode
4955 //   + 名前: 便利な命名法である /usr/rtm/xv6/fs.c のようなパス
4956 //
4957 // このファイルには低レベルのファイルシステム操作関数が含まれている。
4958 // ( 高レベルの )システムコールの実装はsysfile.cにある。
4959
4960 #include "types.h"
4961 #include "defs.h"
4962 #include "param.h"
4963 #include "stat.h"
4964 #include "mmu.h"
4965 #include "proc.h"
4966 #include "spinlock.h"
4967 #include "sleeplock.h"
4968 #include "fs.h"
4969 #include "buf.h"
4970 #include "file.h"
4971
4972 #define min(a, b) ((a) < (b) ? (a) : (b))
4973 static void itrunc(struct inode*);
4974 // ディスク装置ごとに１つスーパーブロックが必要であるが、このOSでは１つしかデバイスを
4975 // 使わない。
4976 struct superblock sb;
4977
4978 // スーパーブロックを読み込む
4979 void
4980 readsb(int dev, struct superblock *sb)
4981 {
4982   struct buf *bp;
4983
4984   bp = bread(dev, 1);
4985   memmove(sb, bp->data, sizeof(*sb));
4986   brelse(bp);
4987 }
4988
4989
4990
4991
4992
4993
4994
4995
4996
4997
4998
4999
```

```
5000 // ブロックを0クリア
5001 static void
5002 bzero(int dev, int bno)
5003 {
5004   struct buf *bp;
5005
5006   bp = bread(dev, bno);
5007   memset(bp->data, 0, BSIZE);
5008   log_write(bp);
5009   brelse(bp);
5010 }
5011
5012 // ブロック
5013
5014 // ディスクブロックを割り当てて0クリア ( ブロック番号を返す )
5015 static uint
5016 balloc(uint dev)
5017 {
5018   int b, bi, m;
5019   struct buf *bp;
5020
5021   bp = 0;
5022   for(b = 0; b < sb.size; b += BPB){
5023     bp = bread(dev, BBLOCK(b, sb));
5024     for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
5025       m = 1 << (bi % 8);
5026       if((bp->data[bi/8] & m) == 0){  // このブロックは空いているか
5027         bp->data[bi/8] |= m;  // ブロックに使用中のマークを付ける。
5028         log_write(bp);
5029         brelse(bp);
5030         bzero(dev, b + bi);
5031         return b + bi;
5032       }
5033     }
5034     brelse(bp);
5035   }
5036   panic("balloc: out of blocks");
5037 }
5038
5039
5040
5041
5042
5043
5044
5045
5046
5047
5048
5049
```



```
5050 // ディスクブロックを解放する
5051 static void
5052 bfree(int dev, uint b)
5053 {
5054   struct buf *bp;
5055   int bi, m;
5056
5057   readsb(dev, &sb);
5058   bp = bread(dev, BBLOCK(b, sb));
5059   bi = b % BPB;
5060   m = 1 << (bi % 8);
5061   if((bp->data[bi/8] & m) == 0)
5062     panic("freeing free block");
5063   bp->data[bi/8] &= ~m;
5064   log_write(bp);
5065   brelse(bp);
5066 }
5067
5068 // inode。
5069 //
5070 // inodeは無名のファイルを一つ記述する。
5071 // inodeディスク構造体はメタデータ ( ファイル種別、サイズ、
5072 // 自身を参照しているリンクの数、ファイルコンテンツを保持
5073 // しているブロックのリスト ) を保持する。
5074 //
5075 // inodeはディスク上のsb.startinodeから連続的に
5076 // 配置されている。各inodeは番号を持っており、
5077 // これはディスク上の位置を示している。
5078 //
5079 // カーネルは使用中のinodeのキャシュをメモリ上に保持しており、
5080 // これにより複数のプロセスが使用するinodeへの同期的アクセスを
5081 // 実現している。キャッシュされたinodeには、ディスクには保存されない
5082 // 記帳情報(book-keeping information)であるip->refとip->validが
5083 // 含まれている。
5084 //
5085 // inodeとそのインメモリコピーは、ファイルシステムのその他のコードが
5086 // 使用できるようになるまでに、次のように状態が変遷する。
5087 //
5088 // * Allocation(割り当て): inodeはその ( ディスク上の ) 種別が
5089 //   0でない場合、割り当てが行われる。
5090 //   ialloc()は割り当てを行い、iput()は参照とリンクのカウントが0に
5091 //   なったら、解放する。
5092 //
5093 // * Referencing in cache(キャッシュ内で参照中): inodeキャッシュのエントリは
5094 //   ip->refが0になると解放される。そうでない場合、ip->refは
5095 //   そのエントリ ( オープンファイルとカレントディレクトリ ) への
5096 //   インメモリポインタの数を監視する。
5097 //   iget()は、キャッシュエントリの検出または作成を行い、
5098 //   refをインクリメントする。iput()はrefをデクリメントする。
5099 //
```

```
5100 // * Valid(有効): inodeキャッシュエントリの情報（種別、サイズなど）は
5101 //   ip->validが1の時にのみ、正しいものである。
5102 //   ilock()はinodeをディスクから読み込み、ip->validをセットする。
5103 //   iput()はip->refが0になった時に、ip->validをクリアする。
5104 //
5105 // * Locked(ロック): ファイルシステムのコードは、inodeを最初にロックした
5106 //   場合にのみ、icode内の情報やそのコンテンツを調べたり、変更したり
5107 //   することができる。
5108 //
5109 // したがって、典型的なコードの流れは次のようになる:
5110 //   ip = iget(dev, inum)
5111 //   ilock(ip)
5112 //   ... ip->xxxのチェックと変更 ...
5113 //   iunlock(ip)
5114 //   iput(ip)
5115 //
5116 // ilock()とiget()が分離されているため、システムコールはinodeへの
5117 // 長期的な参照を得る（ファイルのオープンなど）ことができる一方で、
5118 // （read()のように）inodeの短期的なロックのみをすることもできる。
5119 // この分離はパス名検索の際のデッドロックや競合の防止にも役立っている。
5120 //
5121 // iget()はip->refをインクリメントするので、inodeはキャッシュに留まり、
5122 // inodeを指すポイントが引き続き有効となる。
5123 //
5124 // 内部ファイルシステム関数の多くは、使用するinodeが呼び出し側(caller)で
5125 // ロックされていることを想定している。これは呼び出し側に多段階アトミック
5126 // 操作を作成するよう仕向けるためである。
5127 //
5128 // icache.lockスピンロックはicacheエントリの割り当てを保護する。
5129 // ip->refはエントリが空きであるか否かを、ip->devとip->inumは
5130 // エントリがどのinodeを保持しているかを示すので、これらフィールドの
5131 // いずれかを使用する際は、icache.lockを取得する必要がある。
5132 //
5133 // ip->lockスリープロックは、inode構造体のref, dev, inum以外のすべての
5134 // フィールドを保護する。inodeのip->valid, ip->size, ip->type
5135 // などの読み書きをする際は、ip->lockを取得する必要がある。
5136
5137 struct {
5138   struct spinlock lock;
5139   struct inode inode[NINODE];
5140 } icache;
5141
5142 void
5143 iinit(int dev)
5144 {
5145   int i = 0;
5146
5147   initlock(&icache.lock, "icache");
5148   for(i = 0; i < NINODE; i++) {
5149     initsleeplock(&icache.inode[i].lock, "inode");
```



```
5150   }
5151
5152   readsb(dev, &sb);
5153   cprintf("sb: size %d nblocks %d ninodes %d nlog %d logstart %d\
5154  inodestart %d bmap start %d\n", sb.size, sb.nblocks,
5155           sb.ninodes, sb.nlog, sb.logstart, sb.inodestart,
5156           sb.bmapstart);
5157 }
5158
5159 static struct inode* iget(uint dev, uint inum);
5160
5161
5162
5163
5164
5165
5166
5167
5168
5169
5170
5171
5172
5173
5174
5175
5176
5177
5178
5179
5180
5181
5182
5183
5184
5185
5186
5187
5188
5189
5190
5191
5192
5193
5194
5195
5196
5197
5198
5199
```

```
5200 // デバイスdevにinodeを割り当てる。
5201 // inodeの種別をtypeとし、割り当て済みのマークを付ける。
5202 // 未ロックだが割り当て済みかつ参照済みのinodeを返す。
5203 struct inode*
5204 ialloc(uint dev, short type)
5205 {
5206   int inum;
5207   struct buf *bp;
5208   struct dinode *dip;
5209
5210   for(inum = 1; inum < sb.ninodes; inum++){
5211     bp = bread(dev, IBLOCK(inum, sb));
5212     dip = (struct dinode*)bp->data + inum%IPB;
5213     if(dip->type == 0){  // 空きinode
5214       memset(dip, 0, sizeof(*dip));
5215       dip->type = type;
5216       log_write(bp);   // デスクに割り当て済みのマークを付ける
5217       brelse(bp);
5218       return iget(dev, inum);
5219     }
5220     brelse(bp);
5221   }
5222   panic("ialloc: no inodes");
5223 }
5224
5225 // 変更したインメモリinodeをディスクにコピーする。
5226 // inodeキャッシュはライトスルーなので、ディスクに存在するinodeの
5227 // 任意のip->xxxフィールドを変更した後に呼び出す必要がある。
5228 // 呼び出し側はip->lockを保持しなければならない。
5229 void
5230 iupdate(struct inode *ip)
5231 {
5232   struct buf *bp;
5233   struct dinode *dip;
5234
5235   bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5236   dip = (struct dinode*)bp->data + ip->inum%IPB;
5237   dip->type = ip->type;
5238   dip->major = ip->major;
5239   dip->minor = ip->minor;
5240   dip->nlink = ip->nlink;
5241   dip->size = ip->size;
5242   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
5243   log_write(bp);
5244   brelse(bp);
5245 }
5246
5247
5248
5249
```



```
5250 // デバイスdev上でinode番号inumを持つinodeを探し、
5251 // そのインメモリコピーを返す。inodeはロックせず、
5252 // ディスクからの読み込みもしない。
5253 static struct inode*
5254 iget(uint dev, uint inum)
5255 {
5256   struct inode *ip, *empty;
5257
5258   acquire(&icache.lock);
5259
5260   // 対象のinodeはすでにキャッシュされているか？
5261   empty = 0;
5262   for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
5263     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
5264       ip->ref++;
5265       release(&icache.lock);
5266       return ip;
5267     }
5268     if(empty == 0 && ip->ref == 0)    // 空のスロットを記憶する
5269       empty = ip;
5270   }
5271
5272   // inodeキャッシュエントリをリサイクルする
5273   if(empty == 0)
5274     panic("iget: no inodes");
5275
5276   ip = empty;
5277   ip->dev = dev;
5278   ip->inum = inum;
5279   ip->ref = 1;
5280   ip->valid = 0;
5281   release(&icache.lock);
5282
5283   return ip;
5284 }
5285
5286 // ipの参照カウントをインクリメントする。
5287 // イディオム ip = idup(ip1) を使えるように、ipを返す。
5288 struct inode*
5289 idup(struct inode *ip)
5290 {
5291   acquire(&icache.lock);
5292   ip->ref++;
5293   release(&icache.lock);
5294   return ip;
5295 }
5296
5297
5298
5299
```

```
5300 // 指定されたinodeをロックする。
5301 // 必要であれば、ディスクからinodeを読み込む。
5302 void
5303 ilock(struct inode *ip)
5304 {
5305   struct buf *bp;
5306   struct dinode *dip;
5307
5308   if(ip == 0 || ip->ref < 1)
5309     panic("ilock");
5310
5311   acquiresleep(&ip->lock);
5312
5313   if(ip->valid == 0){
5314     bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5315     dip = (struct dinode*)bp->data + ip->inum%IPB;
5316     ip->type = dip->type;
5317     ip->major = dip->major;
5318     ip->minor = dip->minor;
5319     ip->nlink = dip->nlink;
5320     ip->size = dip->size;
5321     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
5322     brelse(bp);
5323     ip->valid = 1;
5324     if(ip->type == 0)
5325       panic("ilock: no type");
5326   }
5327 }
5328
5329 // 指定されたinodeのロックを外す
5330 void
5331 iunlock(struct inode *ip)
5332 {
5333   if(ip == 0 || !holdingsleep(&ip->lock) || ip->ref < 1)
5334     panic("iunlock");
5335
5336   releasesleep(&ip->lock);
5337 }
5338
5339
5340
5341
5342
5343
5344
5345
5346
5347
5348
5349
```



```
5350 // インメモリinodeへの参照をデクリメントする。
5351 // それが最後の参照だった場合、そのinodeキャッシュエントリは
5352 // リサイクル可能になる。
5353 // それが最後の参照で、そのinodeへのリンクがない場合、
5354 // ディスク上のinode ( とそのコンテンツ ) を解放する。
5355 // inodeを解放する場合に備えて、iput()の呼び出しは常に
5356 // トランザクション内でなければならない。
5357 void
5358 iput(struct inode *ip)
5359 {
5360   acquiresleep(&ip->lock);
5361   if(ip->valid && ip->nlink == 0){
5362     acquire(&icache.lock);
5363     int r = ip->ref;
5364     release(&icache.lock);
5365     if(r == 1){
5366       // inodeはリンクがなく、他に参照もない: 切り詰めて解放する
5367       itrunc(ip);
5368       ip->type = 0;
5369       iupdate(ip);
5370       ip->valid = 0;
5371     }
5372   }
5373   releasesleep(&ip->lock);
5374
5375   acquire(&icache.lock);
5376   ip->ref--;
5377   release(&icache.lock);
5378 }
5379
5380 // 一般的なイディオム: unlockしてputする
5381 void
5382 iunlockput(struct inode *ip)
5383 {
5384   iunlock(ip);
5385   iput(ip);
5386 }
5387
5388
5389
5390
5391
5392
5393
5394
5395
5396
5397
5398
5399
```

```
5400 // inodeのコンテンツ
5401 //
5402 // 各inodeに関連するコンテンツ（データ）はディスクのブロックに
5403 // 格納される。最初のNDIRECT個のブロック番号は
5404 // ip->addrs[]に記録される。次のNINDIRECT個のブロックは
5405 // ip->addrs[NDIRECT]のブロックに記録される。.
5406
5407 // inode ipのn番目のブロックのディスクブロックアドレスを返す。
5408 // そのようなブロックが存在しない場合、bmapはブロックを割り当てる。
5409 static uint
5410 bmap(struct inode *ip, uint bn)
5411 {
5412   uint addr, *a;
5413   struct buf *bp;
5414
5415   if(bn < NDIRECT){
5416     if((addr = ip->addrs[bn]) == 0)
5417       ip->addrs[bn] = addr = balloc(ip->dev);
5418     return addr;
5419   }
5420   bn -= NDIRECT;
5421
5422   if(bn < NINDIRECT){
5423     // 間接ブロックをロードする。必要であれば割り当てる。
5424     if((addr = ip->addrs[NDIRECT]) == 0)
5425       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5426     bp = bread(ip->dev, addr);
5427     a = (uint*)bp->data;
5428     if((addr = a[bn]) == 0){
5429       a[bn] = addr = balloc(ip->dev);
5430       log_write(bp);
5431     }
5432     brelse(bp);
5433     return addr;
5434   }
5435
5436   panic("bmap: out of range");
5437 }
5438
5439
5440
5441
5442
5443
5444
5445
5446
5447
5448
5449
```



```
5450 // inodeを切り詰める（コンテンツを破棄する）。
5451 // そのinodeへのリンクがなく（参照するディレクトリ
5452 // エントリがない）、かつ、そのinodeへのインメモリ参照が
5453 // ない（オープンされたファイルでないか、カレントディレクトリ
5454 // でない）場合にのみ呼び出される。
5455 static void
5456 itrunc(struct inode *ip)
5457 {
5458   int i, j;
5459   struct buf *bp;
5460   uint *a;
5461
5462   for(i = 0; i < NDIRECT; i++){
5463     if(ip->addrs[i]){
5464       bfree(ip->dev, ip->addrs[i]);
5465       ip->addrs[i] = 0;
5466     }
5467   }
5468
5469   if(ip->addrs[NDIRECT]){
5470     bp = bread(ip->dev, ip->addrs[NDIRECT]);
5471     a = (uint*)bp->data;
5472     for(j = 0; j < NINDIRECT; j++){
5473       if(a[j])
5474         bfree(ip->dev, a[j]);
5475     }
5476     brelse(bp);
5477     bfree(ip->dev, ip->addrs[NDIRECT]);
5478     ip->addrs[NDIRECT] = 0;
5479   }
5480
5481   ip->size = 0;
5482   iupdate(ip);
5483 }
5484
5485 // inodeからstat情報をコピーする。
5486 // 呼び出し側でip->lockを取得しなければならない。
5487 void
5488 stati(struct inode *ip, struct stat *st)
5489 {
5490   st->dev = ip->dev;
5491   st->ino = ip->inum;
5492   st->type = ip->type;
5493   st->nlink = ip->nlink;
5494   st->size = ip->size;
5495 }
5496
5497
5498
5499
```

```
// inodeからデータを読み込む。
// 呼び出し側でip->lockを取得しなければならない。
int
readi(struct inode *ip, char *dst, uint off, uint n)
{
  uint tot, m;
  struct buf *bp;

  if(ip->type == T_DEV){
    if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
      return -1;
    return devsw[ip->major].read(ip, dst, n);
  }

  if(off > ip->size || off + n < off)
    return -1;
  if(off + n > ip->size)
    n = ip->size - off;

  for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
    bp = bread(ip->dev, bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    memmove(dst, bp->data + off%BSIZE, m);
    brelse(bp);
  }
  return n;
}
```



```
// データをinodeに書き込む。
// 呼び出し側でip->lockを取得しなければならない。
int
writei(struct inode *ip, char *src, uint off, uint n)
{
  uint tot, m;
  struct buf *bp;

  if(ip->type == T_DEV){
    if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
      return -1;
    return devsw[ip->major].write(ip, src, n);
  }

  if(off > ip->size || off + n < off)
    return -1;
  if(off + n > MAXFILE*BSIZE)
    return -1;

  for(tot=0; tot<n; tot+=m, off+=m, src+=m){
    bp = bread(ip->dev, bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    memmove(bp->data + off%BSIZE, src, m);
    log_write(bp);
    brelse(bp);
  }

  if(n > 0 && off > ip->size){
    ip->size = off;
    iupdate(ip);
  }
  return n;
}
```

```
5600 // ディレクトリ
5601
5602 int
5603 namecmp(const char *s, const char *t)
5604 {
5605   return strncmp(s, t, DIRSIZ);
5606 }
5607
5608 // ディレクトリでディレクトリエントリを検索する。
5609 // 見つかった場合、エントリのバイトオフセットを *poff に設定する。
5610 struct inode*
5611 dirlookup(struct inode *dp, char *name, uint *poff)
5612 {
5613   uint off, inum;
5614   struct dirent de;
5615
5616   if(dp->type != T_DIR)          // dpがディレクトリでない
5617     panic("dirlookup not DIR");
5618
5619   for(off = 0; off < dp->size; off += sizeof(de)){
5620     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5621       panic("dirlookup read");
5622     if(de.inum == 0)
5623       continue;
5624     if(namecmp(name, de.name) == 0){
5625       // エントリがパス要素に一致
5626       if(poff)
5627         *poff = off;
5628       inum = de.inum;
5629       return iget(dp->dev, inum);
5630     }
5631   }
5632
5633   return 0;
5634 }
5635
5636
5637
5638
5639
5640
5641
5642
5643
5644
5645
5646
5647
5648
5649
```



```
5650 // 新しいディレクトリエントリ ( 名前、inum ) をディレクトリdpに書き込む。
5651 int
5652 dirlink(struct inode *dp, char *name, uint inum)
5653 {
5654   int off;
5655   struct dirent de;
5656   struct inode *ip;
5657
5658   // 名前が存在しないかチェックする。
5659   if((ip = dirlookup(dp, name, 0)) != 0){  // 存在した
5660     iput(ip);                              // dirlookupでref++しているのでref--
5661     return -1;
5662   }
5663
5664   // 空のdirentを探す。
5665   for(off = 0; off < dp->size; off += sizeof(de)){
5666     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5667       panic("dirlink read");
5668     if(de.inum == 0)
5669       break;
5670   }
5671
5672   strncpy(de.name, name, DIRSIZ);
5673   de.inum = inum;
5674   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5675     panic("dirlink");
5676
5677   return 0;
5678 }
5679
5680
5681
5682
5683
5684
5685
5686
5687
5688
5689
5690
5691
5692
5693
5694
5695
5696
5697
5698
5699
```

```
5700 // パス
5701
5702 // pathから次のパス要素をnameにコピーする。
5703 // コピーした要素の次の要素へのポインタを返す。
5704 // 呼び出し側が *path=='\0' をチェックして、nameが最終要素であるか
5705 // 否かを判断できるように、返されるパスには先頭にスラッシュを付けない。
5706 // 取り除く名前がなかった場合は、0 を返す。
5707 //
5708 // 例:
5709 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
5710 //   skipelem("///a//bb", name) = "bb", setting name = "a"
5711 //   skipelem("a", name) = "", setting name = "a"
5712 //   skipelem("", name) = skipelem("////", name) = 0
5713 //
5714 static char*
5715 skipelem(char *path, char *name)
5716 {
5717   char *s;
5718   int len;
5719
5720   while(*path == '/')
5721     path++;
5722   if(*path == 0)
5723     return 0;
5724   s = path;
5725   while(*path != '/' && *path != 0)
5726     path++;
5727   len = path - s;
5728   if(len >= DIRSIZ)
5729     memmove(name, s, DIRSIZ);
5730   else {
5731     memmove(name, s, len);
5732     name[len] = 0;
5733   }
5734   while(*path == '/')
5735     path++;
5736   return path;
5737 }
5738
5739
5740
5741
5742
5743
5744
5745
5746
5747
5748
5749
```



```
5750 // パス名を検索して、そのinodeを返す
5751 // nameiparent != 0の場合は、親のinodeを返し、最終パス要素をnameにコピーする。
5752 // nameはDIRSIZ(14)バイトが確保されていなければならない。
5753 // iput()を呼び出すので、トランザクションの内側で呼び出す必要がある。
5754 static struct inode*
5755 namex(char *path, int nameiparent, char *name)
5756 {
5757   struct inode *ip, *next;
5758
5759   if(*path == '/')
5760     ip = iget(ROOTDEV, ROOTINO);
5761   else
5762     ip = idup(myproc()->cwd);
5763
5764   while((path = skipelem(path, name)) != 0){
5765     ilock(ip);
5766     if(ip->type != T_DIR){
5767       iunlockput(ip);
5768       return 0;
5769     }
5770     if(nameiparent && *path == '\0'){
5771       // 1レベル前で処理を停止
5772       iunlock(ip);
5773       return ip;
5774     }
5775     if((next = dirlookup(ip, name, 0)) == 0){
5776       iunlockput(ip);
5777       return 0;
5778     }
5779     iunlockput(ip);
5780     ip = next;
5781   }
5782   if(nameiparent){
5783     iput(ip);
5784     return 0;
5785   }
5786   return ip;
5787 }
5788
5789 struct inode*
5790 namei(char *path)
5791 {
5792   char name[DIRSIZ];
5793   return namex(path, 0, name);
5794 }
5795
5796
5797
5798
5799
```

```
5800 struct inode*
5801 nameiparent(char *path, char *name)
5802 {
5803   return namex(path, 1, name);
5804 }
5805
5806
5807
5808
5809
5810
5811
5812
5813
5814
5815
5816
5817
5818
5819
5820
5821
5822
5823
5824
5825
5826
5827
5828
5829
5830
5831
5832
5833
5834
5835
5836
5837
5838
5839
5840
5841
5842
5843
5844
5845
5846
5847
5848
5849
```



```
5850 //
5851 // ファイル記述子
5852 //
5853
5854 #include "types.h"
5855 #include "defs.h"
5856 #include "param.h"
5857 #include "fs.h"
5858 #include "spinlock.h"
5859 #include "sleeplock.h"
5860 #include "file.h"
5861
5862 struct devsw devsw[NDEV];
5863 struct {
5864   struct spinlock lock;
5865   struct file file[NFILE];
5866 } ftable;
5867
5868 void
5869 fileinit(void)
5870 {
5871   initlock(&ftable.lock, "ftable");
5872 }
5873
5874 // ファイル構造体を割り当てる。
5875 struct file*
5876 filealloc(void)
5877 {
5878   struct file *f;
5879
5880   acquire(&ftable.lock);
5881   for(f = ftable.file; f < ftable.file + NFILE; f++){
5882     if(f->ref == 0){
5883       f->ref = 1;
5884       release(&ftable.lock);
5885       return f;
5886     }
5887   }
5888   release(&ftable.lock);
5889   return 0;
5890 }
5891
5892
5893
5894
5895
5896
5897
5898
5899
```

```
// ファイル f の参照カウントをインクリメントする。
struct file*
filedup(struct file *f)
{
  acquire(&ftable.lock);
  if(f->ref < 1)
    panic("filedup");
  f->ref++;
  release(&ftable.lock);
  return f;
}

// ファイル f を閉じる ( 参照カウントをデクリメントして、0になったら閉じる ) 。
void
fileclose(struct file *f)
{
  struct file ff;

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

  if(ff.type == FD_PIPE)
    pipeclose(ff.pipe, ff.writable);
  else if(ff.type == FD_INODE){
    begin_op();
    iput(ff.ip);
    end_op();
  }
}
```



```
// ファイル f に関するメタデータを取得する。
int
filestat(struct file *f, struct stat *st)
{
  if(f->type == FD_INODE){
    ilock(f->ip);
    stati(f->ip, st);
    iunlock(f->ip);
    return 0;
  }
  return -1;
}

// ファイル f から読み込む。
int
fileread(struct file *f, char *addr, int n)
{
  int r;

  if(f->readable == 0)
    return -1;
  if(f->type == FD_PIPE)
    return piperead(f->pipe, addr, n);
  if(f->type == FD_INODE){
    ilock(f->ip);
    if((r = readi(f->ip, addr, f->off, n)) > 0)
      f->off += r;
    iunlock(f->ip);
    return r;
  }
  panic("fileread");
}
```

```
6000 // ファイル f に書き込む。
6001 int
6002 filewrite(struct file *f, char *addr, int n)
6003 {
6004   int r;
6005
6006   if(f->writable == 0)
6007     return -1;
6008   if(f->type == FD_PIPE)
6009     return pipewrite(f->pipe, addr, n);
6010   if(f->type == FD_INODE){
6011     // 最大ログトランザクションサイズを超えないように
6012     // 数ブロックずつ書き込む。関係するのは、inode, 間接ブロック、
6013     // 割り当てブロック、非アライン書き込み調整用の
6014     // 2ブロック。
6015     // writei()はコンソールなどのデバイスに書き込む場合があるので
6016     // これは本当に低速の部類に入る。
6017     int max = ((MAXOPBLOCKS-1-1-2) / 2) * 512;
6018     int i = 0;
6019     while(i < n){
6020       int n1 = n - i;
6021       if(n1 > max)
6022         n1 = max;
6023
6024       begin_op();
6025       ilock(f->ip);
6026       if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
6027         f->off += r;
6028       iunlock(f->ip);
6029       end_op();
6030
6031       if(r < 0)
6032         break;
6033       if(r != n1)
6034         panic("short filewrite");
6035       i += r;
6036     }
6037     return i == n ? n : -1;
6038   }
6039   panic("filewrite");
6040 }
6041
6042
6043
6044
6045
6046
6047
6048
6049
```



```
6050 //
6051 // ファイルシステム関連のシステムコール。
6052 // ユーザのコードは信用できないし、file.cとfs.cを呼び出すので
6053 // ほとんどの関数で引数をチェックする。
6054 //
6055
6056 #include "types.h"
6057 #include "defs.h"
6058 #include "param.h"
6059 #include "stat.h"
6060 #include "mmu.h"
6061 #include "proc.h"
6062 #include "fs.h"
6063 #include "spinlock.h"
6064 #include "sleeplock.h"
6065 #include "file.h"
6066 #include "fcntl.h"
6067
6068 // システムコールのn番目のワードサイズの引数をファイル記述子として取り出し、
6069 // その記述子と対応するfile構造体を返す。
6070 static int
6071 argfd(int n, int *pfd, struct file **pf)
6072 {
6073   int fd;
6074   struct file *f;
6075
6076   if(argint(n, &fd) < 0)
6077     return -1;
6078   if(fd < 0 ¦¦ fd >= NOFILE ¦¦ (f=myproc()->ofile[fd]) == 0)
6079     return -1;
6080   if(pfd)
6081     *pfd = fd;
6082   if(pf)
6083     *pf = f;
6084   return 0;
6085 }
6086
6087
6088
6089
6090
6091
6092
6093
6094
6095
6096
6097
6098
6099
```

```
6100 // 指定したファイルにファイル記述子を割り当てる。
6101 // 成功したら、呼び出し側が指定したファイル参照をファイル記述子に設定する。
6102 static int
6103 fdalloc(struct file *f)
6104 {
6105   int fd;
6106   struct proc *curproc = myproc();
6107
6108   for(fd = 0; fd < NOFILE; fd++){
6109     if(curproc->ofile[fd] == 0){
6110       curproc->ofile[fd] = f;
6111       return fd;
6112     }
6113   }
6114   return -1;
6115 }
6116
6117 int
6118 sys_dup(void)
6119 {
6120   struct file *f;
6121   int fd;
6122
6123   if(argfd(0, 0, &f) < 0)  // 引数1で指定されたfdに対応するファイルを取得
6124     return -1;
6125   if((fd=fdalloc(f)) < 0)  // 取得したファイルに別のfdを与える
6126     return -1;
6127   filedup(f);              // 取得したファイルの参照カウントをインクリメント
6128   return fd;               // 新たに取得したfdを返す
6129 }
6130
6131 int
6132 sys_read(void)
6133 {
6134   struct file *f;         // 引数1でfdとして指定
6135   int n;                  // 引数3で指定
6136   char *p;                // 引数2で指定（ポインタ）
6137
6138   if(argfd(0, 0, &f) < 0 ¦¦ argint(2, &n) < 0 ¦¦ argptr(1, &p, n) < 0)
6139     return -1;
6140   return fileread(f, p, n);
6141 }
6142
6143
6144
6145
6146
6147
6148
6149
```



```
6150 int
6151 sys_write(void)
6152 {
6153   struct file *f;         // 引数1でfdとして指定
6154   int n;                  // 引数3で指定
6155   char *p;                // 引数2で指定（ポインタ）
6156
6157   if(argfd(0, 0, &f) < 0 ¦¦ argint(2, &n) < 0 ¦¦ argptr(1, &p, n) < 0)
6158     return -1;
6159   return filewrite(f, p, n);
6160 }
6161
6162 int
6163 sys_close(void)
6164 {
6165   int fd;
6166   struct file *f;
6167
6168   if(argfd(0, &fd, &f) < 0)
6169     return -1;
6170   myproc()->ofile[fd] = 0;
6171   fileclose(f);
6172   return 0;
6173 }
6174
6175 int
6176 sys_fstat(void)
6177 {
6178   struct file *f;
6179   struct stat *st;
6180
6181   if(argfd(0, 0, &f) < 0 ¦¦ argptr(1, (void*)&st, sizeof(*st)) < 0)
6182     return -1;
6183   return filestat(f, st);
6184 }
6185
6186
6187
6188
6189
6190
6191
6192
6193
6194
6195
6196
6197
6198
6199
```

```
6200 // old と同じinodeへのリンクとして新しパスを作成する。
6201 int
6202 sys_link(void)
6203 {
6204   char name[DIRSIZ], *new, *old;
6205   struct inode *dp, *ip;
6206
6207   if(argstr(0, &old) < 0 ¦¦ argstr(1, &new) < 0)
6208     return -1;
6209
6210   begin_op();
6211   if((ip = namei(old)) == 0){
6212     end_op();
6213     return -1;
6214   }
6215
6216   ilock(ip);
6217   if(ip->type == T_DIR){
6218     iunlockput(ip);
6219     end_op();
6220     return -1;
6221   }
6222
6223   ip->nlink++;
6224   iupdate(ip);
6225   iunlock(ip);
6226
6227   if((dp = nameiparent(new, name)) == 0)
6228     goto bad;
6229   ilock(dp);
6230   if(dp->dev != ip->dev ¦¦ dirlink(dp, name, ip->inum) < 0){
6231     iunlockput(dp);
6232     goto bad;
6233   }
6234   iunlockput(dp);
6235   iput(ip);
6236
6237   end_op();
6238
6239   return 0;
6240
6241 bad:
6242   ilock(ip);
6243   ip->nlink--;
6244   iupdate(ip);
6245   iunlockput(ip);
6246   end_op();
6247   return -1;
6248 }
6249
```



```
6250 // ディレクトリdpは"."と".."を除いて、空であるか?
6251 static int
6252 isdirempty(struct inode *dp)
6253 {
6254   int off;
6255   struct dirent de;
6256
6257   for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
6258     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6259       panic("isdirempty: readi");
6260     if(de.inum != 0)
6261       return 0;
6262   }
6263   return 1;
6264 }
6265
6266
6267
6268
6269
6270
6271
6272
6273
6274
6275
6276
6277
6278
6279
6280
6281
6282
6283
6284
6285
6286
6287
6288
6289
6290
6291
6292
6293
6294
6295
6296
6297
6298
6299
```

```
6300 int
6301 sys_unlink(void)
6302 {
6303   struct inode *ip, *dp;
6304   struct dirent de;
6305   char name[DIRSIZ], *path;
6306   uint off;
6307
6308   if(argstr(0, &path) < 0)
6309     return -1;
6310
6311   begin_op();
6312   if((dp = nameiparent(path, name)) == 0){
6313     end_op();
6314     return -1;
6315   }
6316
6317   ilock(dp);
6318
6319   // "."と".."はunlinkできない。
6320   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
6321     goto bad;
6322
6323   if((ip = dirlookup(dp, name, &off)) == 0)  // 指定のpathは存在しない
6324     goto bad;
6325   ilock(ip);
6326
6327   if(ip->nlink < 1)
6328     panic("unlink: nlink < 1");
6329   if(ip->type == T_DIR && !isdirempty(ip)){  // 指定ディレクトリにファイルあり
6330     iunlockput(ip);
6331     goto bad;
6332   }
6333
6334   memset(&de, 0, sizeof(de));
6335   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6336     panic("unlink: writei");
6337   if(ip->type == T_DIR){
6338     dp->nlink--;
6339     iupdate(dp);
6340   }
6341   iunlockput(dp);
6342
6343   ip->nlink--;
6344   iupdate(ip);
6345   iunlockput(ip);
6346
6347   end_op();
6348
6349   return 0;
```



```
6350 bad:
6351   iunlockput(dp);
6352   end_op();
6353   return -1;
6354 }
6355
6356 static struct inode*
6357 create(char *path, short type, short major, short minor)
6358 {
6359   uint off;
6360   struct inode *ip, *dp;
6361   char name[DIRSIZ];
6362
6363   if((dp = nameiparent(path, name)) == 0)
6364     return 0;
6365   ilock(dp);
6366
6367   if((ip = dirlookup(dp, name, &off)) != 0){
6368     iunlockput(dp);
6369     ilock(ip);
6370     if(type == T_FILE && ip->type == T_FILE)  // 同名のファイルあり
6371       return ip;
6372     iunlockput(ip);
6373     return 0;
6374   }
6375
6376   if((ip = ialloc(dp->dev, type)) == 0)  // 新規inode割り当て
6377     panic("create: ialloc");
6378
6379   ilock(ip);
6380   ip->major = major;
6381   ip->minor = minor;
6382   ip->nlink = 1;
6383   iupdate(ip);
6384
6385   if(type == T_DIR){  // "."と".."エントリを作成する
6386     dp->nlink++;  // ".."エントリ
6387     iupdate(dp);
6388     // "."では、循環参照カウントを避けるために、ip->nlink++をしない。
6389     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
6390       panic("create dots");
6391   }
6392
6393   if(dirlink(dp, name, ip->inum) < 0)
6394     panic("create: dirlink");
6395
6396   iunlockput(dp);
6397
6398   return ip;
6399 }
```

```
6400 int
6401 sys_open(void)
6402 {
6403   char *path;
6404   int fd, omode;
6405   struct file *f;
6406   struct inode *ip;
6407
6408   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
6409     return -1;
6410
6411   begin_op();
6412
6413   if(omode & O_CREATE){
6414     ip = create(path, T_FILE, 0, 0);
6415     if(ip == 0){
6416       end_op();
6417       return -1;
6418     }
6419   } else {
6420     if((ip = namei(path)) == 0){
6421       end_op();
6422       return -1;
6423     }
6424     ilock(ip);
6425     if(ip->type == T_DIR && omode != O_RDONLY){
6426       iunlockput(ip);
6427       end_op();
6428       return -1;
6429     }
6430   }
6431
6432   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
6433     if(f)
6434       fileclose(f);
6435     iunlockput(ip);
6436     end_op();
6437     return -1;
6438   }
6439   iunlock(ip);
6440   end_op();
6441
6442   f->type = FD_INODE;
6443   f->ip = ip;
6444   f->off = 0;
6445   f->readable = !(omode & O_WRONLY);
6446   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
6447   return fd;
6448 }
6449
```



```
6450 int
6451 sys_mkdir(void)
6452 {
6453   char *path;
6454   struct inode *ip;
6455
6456   begin_op();
6457   if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
6458     end_op();
6459     return -1;
6460   }
6461   iunlockput(ip);
6462   end_op();
6463   return 0;
6464 }
6465
6466 int
6467 sys_mknod(void)
6468 {
6469   struct inode *ip;
6470   char *path;
6471   int major, minor;
6472
6473   begin_op();
6474   if((argstr(0, &path)) < 0 ||
6475      argint(1, &major) < 0 ||
6476      argint(2, &minor) < 0 ||
6477      (ip = create(path, T_DEV, major, minor)) == 0){
6478     end_op();
6479     return -1;
6480   }
6481   iunlockput(ip);
6482   end_op();
6483   return 0;
6484 }
6485
6486
6487
6488
6489
6490
6491
6492
6493
6494
6495
6496
6497
6498
6499
```

```
6500 int
6501 sys_chdir(void)
6502 {
6503   char *path;
6504   struct inode *ip;
6505   struct proc *curproc = myproc();
6506
6507   begin_op();
6508   if(argstr(0, &path) < 0 ¦¦ (ip = namei(path)) == 0){
6509     end_op();
6510     return -1;
6511   }
6512   ilock(ip);
6513   if(ip->type != T_DIR){
6514     iunlockput(ip);
6515     end_op();
6516     return -1;
6517   }
6518   iunlock(ip);
6519   iput(curproc->cwd);
6520   end_op();
6521   curproc->cwd = ip;
6522   return 0;
6523 }
6524
6525 int
6526 sys_exec(void)
6527 {
6528   char *path, *argv[MAXARG];
6529   int i;
6530   uint uargv, uarg;
6531
6532   if(argstr(0, &path) < 0 ¦¦ argint(1, (int*)&uargv) < 0){
6533     return -1;
6534   }
6535   memset(argv, 0, sizeof(argv));
6536   for(i=0;; i++){
6537     if(i >= NELEM(argv))
6538       return -1;
6539     if(fetchint(uargv+4*i, (int*)&uarg) < 0)
6540       return -1;
6541     if(uarg == 0){
6542       argv[i] = 0;
6543       break;
6544     }
6545     if(fetchstr(uarg, &argv[i]) < 0)
6546       return -1;
6547   }
6548   return exec(path, argv);
6549 }
```



```
6550 int
6551 sys_pipe(void)
6552 {
6553   int *fd;
6554   struct file *rf, *wf;
6555   int fd0, fd1;
6556
6557   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
6558     return -1;
6559   if(pipealloc(&rf, &wf) < 0)
6560     return -1;
6561   fd0 = -1;
6562   if((fd0 = fdalloc(rf)) < 0 ¦¦ (fd1 = fdalloc(wf)) < 0){
6563     if(fd0 >= 0)
6564       myproc()->ofile[fd0] = 0;
6565     fileclose(rf);
6566     fileclose(wf);
6567     return -1;
6568   }
6569   fd[0] = fd0;
6570   fd[1] = fd1;
6571   return 0;
6572 }
6573
6574
6575
6576
6577
6578
6579
6580
6581
6582
6583
6584
6585
6586
6587
6588
6589
6590
6591
6592
6593
6594
6595
6596
6597
6598
6599
```

```
6600 #include "types.h"
6601 #include "param.h"
6602 #include "memlayout.h"
6603 #include "mmu.h"
6604 #include "proc.h"
6605 #include "defs.h"
6606 #include "x86.h"
6607 #include "elf.h"
6608
6609 int
6610 exec(char *path, char **argv)
6611 {
6612   char *s, *last;
6613   int i, off;
6614   uint argc, sz, sp, ustack[3+MAXARG+1];
6615   struct elfhdr elf;
6616   struct inode *ip;
6617   struct proghdr ph;
6618   pde_t *pgdir, *oldpgdir;
6619   struct proc *curproc = myproc();
6620
6621   begin_op();
6622
6623   if((ip = namei(path)) == 0){
6624     end_op();
6625     cprintf("exec: fail\n");
6626     return -1;
6627   }
6628   ilock(ip);
6629   pgdir = 0;
6630
6631   // ELFヘッダーをチェックする
6632   if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
6633     goto bad;
6634   if(elf.magic != ELF_MAGIC)
6635     goto bad;
6636
6637   if((pgdir = setupkvm()) == 0)
6638     goto bad;
6639
6640   // プログラムをメモリにロードする。
6641   sz = 0;
6642   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6643     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6644       goto bad;
6645     if(ph.type != ELF_PROG_LOAD)
6646       continue;
6647     if(ph.memsz < ph.filesz)
6648       goto bad;
6649     if(ph.vaddr + ph.memsz < ph.vaddr)
```



```
6650       goto bad;
6651     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6652       goto bad;
6653     if(ph.vaddr % PGSIZE != 0)
6654       goto bad;
6655     if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
6656       goto bad;
6657   }
6658   iunlockput(ip);
6659   end_op();
6660   ip = 0;
6661
6662   // 次のページ境界に2ページ割り当てる。
6663   // 最初のページをアクセス不可にする。2番めのページをユーザスタックとして使用する。
6664   sz = PGROUNDUP(sz);
6665   if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
6666     goto bad;
6667   clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
6668   sp = sz;
6669
6670   // 引数文字列をプッシュし、ustackの残りの要素を用意する。
6671   for(argc = 0; argv[argc]; argc++) {
6672     if(argc >= MAXARG)
6673       goto bad;
6674     sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
6675     if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
6676       goto bad;
6677     ustack[3+argc] = sp;
6678   }
6679   ustack[3+argc] = 0;
6680
6681   ustack[0] = 0xffffffff;  // フェイク復帰PC
6682   ustack[1] = argc;
6683   ustack[2] = sp - (argc+1)*4;  // argvポインタ
6684
6685   sp -= (3+argc+1) * 4;
6686   if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
6687     goto bad;
6688
6689   // デバッグ用にプログラム名を保存する。
6690   for(last=s=path; *s; s++)
6691     if(*s == '/')
6692       last = s+1;
6693   safestrcpy(curproc->name, last, sizeof(curproc->name));
6694
6695   // ユーザイメージにコミットする。
6696   oldpgdir = curproc->pgdir;
6697   curproc->pgdir = pgdir;
6698   curproc->sz = sz;
6699   curproc->tf->eip = elf.entry;  // main
```

```
6700   curproc->tf->esp = sp;
6701   switchuvm(curproc);
6702   freevm(oldpgdir);
6703   return 0;
6704
6705  bad:
6706   if(pgdir)
6707     freevm(pgdir);
6708   if(ip){
6709     iunlockput(ip);
6710     end_op();
6711   }
6712   return -1;
6713 }
6714
6715
6716
6717
6718
6719
6720
6721
6722
6723
6724
6725
6726
6727
6728
6729
6730
6731
6732
6733
6734
6735
6736
6737
6738
6739
6740
6741
6742
6743
6744
6745
6746
6747
6748
6749
```



```
6750 #include "types.h"
6751 #include "defs.h"
6752 #include "param.h"
6753 #include "mmu.h"
6754 #include "proc.h"
6755 #include "fs.h"
6756 #include "spinlock.h"
6757 #include "sleeplock.h"
6758 #include "file.h"
6759
6760 #define PIPESIZE 512
6761
6762 struct pipe {
6763   struct spinlock lock;
6764   char data[PIPESIZE];
6765   uint nread;     // 読み込まれたバイト数
6766   uint nwrite;    // 書き込まれたバイト数
6767   int readopen;   // read fdはまだオープンされている
6768   int writeopen;  // write fdはまだオープンされている
6769 };
6770
6771 int
6772 pipealloc(struct file **f0, struct file **f1)
6773 {
6774   struct pipe *p;
6775
6776   p = 0;
6777   *f0 = *f1 = 0;
6778   if((*f0 = filealloc()) == 0 ¦¦ (*f1 = filealloc()) == 0)
6779     goto bad;
6780   if((p = (struct pipe*)kalloc()) == 0)
6781     goto bad;
6782   p->readopen = 1;
6783   p->writeopen = 1;
6784   p->nwrite = 0;
6785   p->nread = 0;
6786   initlock(&p->lock, "pipe");
6787   (*f0)->type = FD_PIPE;
6788   (*f0)->readable = 1;
6789   (*f0)->writable = 0;
6790   (*f0)->pipe = p;
6791   (*f1)->type = FD_PIPE;
6792   (*f1)->readable = 0;
6793   (*f1)->writable = 1;
6794   (*f1)->pipe = p;
6795   return 0;
6796
6797
6798
6799
```

```
6800  bad:
6801    if(p)
6802      kfree((char*)p);
6803    if(*f0)
6804      fileclose(*f0);
6805    if(*f1)
6806      fileclose(*f1);
6807    return -1;
6808  }
6809
6810  void
6811  pipeclose(struct pipe *p, int writable)
6812  {
6813    acquire(&p->lock);
6814    if(writable){
6815      p->writeopen = 0;
6816      wakeup(&p->nread);
6817    } else {
6818      p->readopen = 0;
6819      wakeup(&p->nwrite);
6820    }
6821    if(p->readopen == 0 && p->writeopen == 0){
6822      release(&p->lock);
6823      kfree((char*)p);
6824    } else
6825      release(&p->lock);
6826  }
6827
6828
6829  int
6830  pipewrite(struct pipe *p, char *addr, int n)
6831  {
6832    int i;
6833
6834    acquire(&p->lock);
6835    for(i = 0; i < n; i++){
6836      while(p->nwrite == p->nread + PIPESIZE){
6837        if(p->readopen == 0 ¦¦ myproc()->killed){
6838          release(&p->lock);
6839          return -1;
6840        }
6841        wakeup(&p->nread);
6842        sleep(&p->nwrite, &p->lock);
6843      }
6844      p->data[p->nwrite++ % PIPESIZE] = addr[i];
6845    }
6846    wakeup(&p->nread);
6847    release(&p->lock);
6848    return n;
6849  }
```



```
6850  int
6851  piperead(struct pipe *p, char *addr, int n)
6852  {
6853    int i;
6854
6855    acquire(&p->lock);
6856    while(p->nread == p->nwrite && p->writeopen){
6857      if(myproc()->killed){
6858        release(&p->lock);
6859        return -1;
6860      }
6861      sleep(&p->nread, &p->lock);
6862    }
6863    for(i = 0; i < n; i++){
6864      if(p->nread == p->nwrite)
6865        break;
6866      addr[i] = p->data[p->nread++ % PIPESIZE];
6867    }
6868    wakeup(&p->nwrite);
6869    release(&p->lock);
6870    return i;
6871  }
6872
6873
6874
6875
6876
6877
6878
6879
6880
6881
6882
6883
6884
6885
6886
6887
6888
6889
6890
6891
6892
6893
6894
6895
6896
6897
6898
6899
```

```
6900 #include "types.h"
6901 #include "x86.h"
6902
6903 void*
6904 memset(void *dst, int c, uint n)
6905 {
6906   if ((int)dst%4 == 0 && n%4 == 0){
6907     c &= 0xFF;
6908     stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
6909   } else
6910     stosb(dst, c, n);
6911   return dst;
6912 }
6913
6914 int
6915 memcmp(const void *v1, const void *v2, uint n)
6916 {
6917   const uchar *s1, *s2;
6918
6919   s1 = v1;
6920   s2 = v2;
6921   while(n-- > 0){
6922     if(*s1 != *s2)
6923       return *s1 - *s2;
6924     s1++, s2++;
6925   }
6926
6927   return 0;
6928 }
6929
6930 void*
6931 memmove(void *dst, const void *src, uint n)
6932 {
6933   const char *s;
6934   char *d;
6935
6936   s = src;
6937   d = dst;
6938   if(s < d && s + n > d){
6939     s += n;
6940     d += n;
6941     while(n-- > 0)
6942       *--d = *--s;
6943   } else
6944     while(n-- > 0)
6945       *d++ = *s++;
6946
6947   return dst;
6948 }
6949
```



```
6950 // memcpyはGCCをなだめるために存在する。memmoveを使用すること。
6951 void*
6952 memcpy(void *dst, const void *src, uint n)
6953 {
6954   return memmove(dst, src, n);
6955 }
6956
6957 int
6958 strncmp(const char *p, const char *q, uint n)
6959 {
6960   while(n > 0 && *p && *p == *q)
6961     n--, p++, q++;
6962   if(n == 0)
6963     return 0;
6964   return (uchar)*p - (uchar)*q;
6965 }
6966
6967 char*
6968 strncpy(char *s, const char *t, int n)
6969 {
6970   char *os;
6971
6972   os = s;
6973   while(n-- > 0 && (*s++ = *t++) != 0)
6974     ;
6975   while(n-- > 0)
6976     *s++ = 0;
6977   return os;
6978 }
6979
6980 // strncpyと同じだが、NULL終端が保証されている。
6981 char*
6982 safestrcpy(char *s, const char *t, int n)
6983 {
6984   char *os;
6985
6986   os = s;
6987   if(n <= 0)
6988     return os;
6989   while(--n > 0 && (*s++ = *t++) != 0)
6990     ;
6991   *s = 0;
6992   return os;
6993 }
6994
6995
6996
6997
6998
6999
```

```
7000 int
7001 strlen(const char *s)
7002 {
7003   int n;
7004
7005   for(n = 0; s[n]; n++)
7006     ;
7007   return n;
7008 }
7009
7010
7011
7012
7013
7014
7015
7016
7017
7018
7019
7020
7021
7022
7023
7024
7025
7026
7027
7028
7029
7030
7031
7032
7033
7034
7035
7036
7037
7038
7039
7040
7041
7042
7043
7044
7045
7046
7047
7048
7049
```



```
7050 // マルチプロセッサ仕様書第1巻を参照 [14]
7051
7052 struct mp {             // 浮動ポインタ
7053   uchar signature[4];           // "_MP_"
7054   void *physaddr;               // MP構成テーブルの物理アドレス
7055   uchar length;                 // 1
7056   uchar specrev;                // [14]
7057   uchar checksum;               // すべてのバイトの合計は0でなければならない
7058   uchar type;                   // MPシステム構成種別
7059   uchar imcrp;
7060   uchar reserved[3];
7061 };
7062
7063 struct mpconf {         // 構成テーブルヘッダ
7064   uchar signature[4];           // "PCMP"
7065   ushort length;                // 総テーブル長
7066   uchar version;                // [14]
7067   uchar checksum;               // すべてのバイトの合計は0でなければならない
7068   uchar product[20];            // 製品ID
7069   uint *oemtable;               // OEM テーブルポインタ
7070   ushort oemlength;             // OEM テーブル長
7071   ushort entry;                 // エントリカウント
7072   uint *lapicaddr;              // ローカルAPICのアドレス
7073   ushort xlength;               // 拡張テーブル長
7074   uchar xchecksum;              // 拡張テーブルチェックサム
7075   uchar reserved;
7076 };
7077
7078 struct mpproc {         // プロセッサテーブルエントリ
7079   uchar type;                   // エントリ種別 (0)
7080   uchar apicid;                 // ローカルAPIC ID
7081   uchar version;                // ローカルAPIC バージョン
7082   uchar flags;                  // CPU フラグ
7083     #define MPBOOT 0x02           // このプロセッサはブートストラッププロセッサ
7084   uchar signature[4];           // CPU シグネチャ
7085   uint feature;                 // CPUID命令のfeatureフラグ
7086   uchar reserved[8];
7087 };
7088
7089 struct mpioapic {       // I/O APIC テーブルエントリ
7090   uchar type;                   // エントリ種別 (2)
7091   uchar apicno;                 // I/O APIC ID
7092   uchar version;                // I/O APIC バージョン
7093   uchar flags;                  // I/O APIC フラグ
7094   uint *addr;                   // I/O APIC アドレス
7095 };
7096
7097
7098
7099
```

```
7100 // テーブルエントリ種別
7101 #define MPPROC    0x00  // プロセッサごとに1つ
7102 #define MPBUS     0x01  // バスごとに1つ
7103 #define MPIOAPIC  0x02  // I/O APICごとに1つ
7104 #define MPIOINTR  0x03  // バス割り込みソースごとに1つ
7105 #define MPLINTR   0x04  // システム割り込みソースごとに1つ
7106
7107
7108
7109
7110
7111
7112
7113
7114
7115
7116
7117
7118
7119
7120
7121
7122
7123
7124
7125
7126
7127
7128
7129
7130
7131
7132
7133
7134
7135
7136
7137
7138
7139
7140
7141
7142
7143
7144
7145
7146
7147
7148
7149
```



```
7150 // Blank page.
7151
7152
7153
7154
7155
7156
7157
7158
7159
7160
7161
7162
7163
7164
7165
7166
7167
7168
7169
7170
7171
7172
7173
7174
7175
7176
7177
7178
7179
7180
7181
7182
7183
7184
7185
7186
7187
7188
7189
7190
7191
7192
7193
7194
7195
7196
7197
7198
7199
```

```
7200 // マルチプロセッササポート
7201 // MP記述構造体をメモリ上で探索する。
7202 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
7203
7204 #include "types.h"
7205 #include "defs.h"
7206 #include "param.h"
7207 #include "memlayout.h"
7208 #include "mp.h"
7209 #include "x86.h"
7210 #include "mmu.h"
7211 #include "proc.h"
7212
7213 struct cpu cpus[NCPU];
7214 int ncpu;
7215 uchar ioapicid;
7216
7217 static uchar
7218 sum(uchar *addr, int len)
7219 {
7220   int i, sum;
7221
7222   sum = 0;
7223   for(i=0; i<len; i++)   // int(32bit)で計算してuchar(8bit)で返す。
7224     sum += addr[i];      // checksumとしては最下位1バイトが0であれば良い
7225   return sum;
7226 }
7227
7228 // addrからlenバイト内でMP構造体を探索する。
7229 static struct mp*
7230 mpsearch1(uint a, int len)
7231 {
7232   uchar *e, *p, *addr;
7233
7234   addr = P2V(a);
7235   e = addr+len;
7236   for(p = addr; p < e; p += sizeof(struct mp))
7237     if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
7238       return (struct mp*)p;
7239   return 0;
7240 }
7241
7242
7243
7244
7245
7246
7247
7248
7249
```



```
7250 // MP浮動ポインタ構造体を探索する。
7251 // 仕様書によれば次の3箇所のいずれかにある。
7252 // 1) EBDAの最初のKB内;
7253 // 2) システムベースメモリの最後のKB内;
7254 // 3) BIOS ROMの0xF0000から0xFFFFFの間。( 訳注: 仕様書には0xF0000とあるのでtypoだろう )
7255 static struct mp*
7256 mpsearch(void)
7257 {
7258   uchar *bda;
7259   uint p;
7260   struct mp *mp;
7261
7262   bda = (uchar *) P2V(0x400);
7263   if((p = ((bda[0x0F]<<8)¦ bda[0x0E]) << 4)){
7264     if((mp = mpsearch1(p, 1024)))
7265       return mp;
7266   } else {
7267     p = ((bda[0x14]<<8)¦bda[0x13])*1024;
7268     if((mp = mpsearch1(p-1024, 1024)))
7269       return mp;
7270   }
7271   return mpsearch1(0xF0000, 0x10000);
7272 }
7273
7274 // MP構成テーブルを探索する。さしあたり、デフォルトの
7275 // 構成 ( physaddr == 0 ) は受け付けない。
7276 // シグネチャが正しいかチェックし、チェックサムを計算する。
7277 // 正しければ、バージョンをチェックする。
7278 // TODO: 拡張テーブルチェックサムをチェックする。
7279 static struct mpconf*
7280 mpconfig(struct mp **pmp)
7281 {
7282   struct mpconf *conf;
7283   struct mp *mp;
7284
7285   if((mp = mpsearch()) == 0 ¦¦ mp->physaddr == 0)
7286     return 0;
7287   conf = (struct mpconf*) P2V((uint) mp->physaddr);
7288   if(memcmp(conf, "PCMP", 4) != 0)
7289     return 0;
7290   if(conf->version != 1 && conf->version != 4)
7291     return 0;
7292   if(sum((uchar*)conf, conf->length) != 0)
7293     return 0;
7294   *pmp = mp;
7295   return conf;
7296 }
7297
7298
7299
```

```
7300 void
7301 mpinit(void)
7302 {
7303   uchar *p, *e;
7304   int ismp;
7305   struct mp *mp;
7306   struct mpconf *conf;
7307   struct mpproc *proc;
7308   struct mpioapic *ioapic;
7309
7310   if((conf = mpconfig(&mp)) == 0)
7311     panic("Expect to run on an SMP");
7312   ismp = 1;
7313   lapic = (uint*)conf->lapicaddr;
7314   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
7315     switch(*p){
7316     case MPPROC:
7317       proc = (struct mpproc*)p;
7318       if(ncpu < NCPU) {
7319         cpus[ncpu].apicid = proc->apicid;  // apicid はncpuとは異なる可能性あり
7320         ncpu++;
7321       }
7322       p += sizeof(struct mpproc);
7323       continue;
7324     case MPIOAPIC:
7325       ioapic = (struct mpioapic*)p;
7326       ioapicid = ioapic->apicno;
7327       p += sizeof(struct mpioapic);
7328       continue;
7329     case MPBUS:
7330     case MPIOINTR:
7331     case MPLINTR:
7332       p += 8;
7333       continue;
7334     default:
7335       ismp = 0;
7336       break;
7337     }
7338   }
7339   if(!ismp)
7340     panic("Didn't find a suitable machine");
7341
7342   if(mp->imcrp){
7343     // BochsはIMCRをサポートしていない。そのため、Bochs上では動かない。
7344     // しかし、実際のマシン上では動くはず。
7345     outb(0x22, 0x70);   // IMCRを選択
7346     outb(0x23, inb(0x23) | 1);  // 外部割り込みをマスクする
7347   }
7348 }
7349
```



```
7350 // ローカルAPICは内部（非I/O）割り込みを管理する。
7351 // Intelプロセッサマニュアル第3巻の8章と付録Cを参照のこと。
7352
7353 #include "param.h"
7354 #include "types.h"
7355 #include "defs.h"
7356 #include "date.h"
7357 #include "memlayout.h"
7358 #include "traps.h"
7359 #include "mmu.h"
7360 #include "x86.h"
7361
7362 // ローカルAPICレジスタ、unit[]のインデックスとして使うために4で割っている。
7363 #define ID      (0x0020/4)   // ID
7364 #define VER     (0x0030/4)   // バージョン
7365 #define TPR     (0x0080/4)   // タスク優先度
7366 #define EOI     (0x00B0/4)   // EOI
7367 #define SVR     (0x00F0/4)   // スプリアス割り込みベクタ
7368   #define ENABLE     0x00000100   // ユニットイネーブル
7369 #define ESR     (0x0280/4)   // エラーステータス
7370 #define ICRLO   (0x0300/4)   // 割り込みコマンド
7371   #define INIT       0x00000500   // INIT/RESET
7372   #define STARTUP    0x00000600   // スタートアップIPI
7373   #define DELIVS     0x00001000   // デリバリステータス
7374   #define ASSERT     0x00004000   // アサート割り込み (vs deassert)
7375   #define DEASSERT   0x00000000
7376   #define LEVEL      0x00008000   // レベルトリガ
7377   #define BCAST      0x00080000   // 自分を含め、すべてのAPICに送信
7378   #define BUSY       0x00001000
7379   #define FIXED      0x00000000
7380 #define ICRHI   (0x0310/4)   // 割り込みコマンド [63:32]
7381 #define TIMER   (0x0320/4)   // ローカルベクタテーブル 0 (TIMER)
7382   #define X1         0x0000000B   // カウントを1で割る
7383   #define PERIODIC   0x00020000   // 定期的
7384 #define PCINT   (0x0340/4)   // 性能モニタリングカウンタLVT
7385 #define LINT0   (0x0350/4)   // ローカルベクタテーブル 1 (LINT0)
7386 #define LINT1   (0x0360/4)   // ローカルベクタテーブル 2 (LINT1)
7387 #define ERROR   (0x0370/4)   // ローカルベクタテーブル 3 (ERROR)
7388   #define MASKED     0x00010000   // 割り込みマスク
7389 #define TICR    (0x0380/4)   // タイマー初期カウント
7390 #define TCCR    (0x0390/4)   // タイマーカレントカウント
7391 #define TDCR    (0x03E0/4)   // タイマー除算設定
7392
7393 volatile uint *lapic;  // mp.cで初期化される
7394
7395
7396
7397
7398
7399
```

```
7400 static void
7401 lapicw(int index, int value)
7402 {
7403   lapic[index] = value;
7404   lapic[ID];  // 読み込むことで、書き込みの完了を待つ
7405 }
7406
7407 void
7408 lapicinit(void)
7409 {
7410   if(!lapic)
7411     return;
7412
7413   // ローカルAPICを有効化; スプリアス割り込みベクタをセットする。
7414   lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
7415
7416   // タイマーはバス周波数でlapic[TICR]から繰り返しカウントダウンし、
7417   // 割り込みを発行する。
7418   // xv6でもっと正確な時間管理をしたいのなら、
7419   // 外部のタイムソースを使ってTICRを補正するとよいだろう。
7420   lapicw(TDCR, X1);
7421   lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
7422   lapicw(TICR, 10000000);
7423
7424   // 論理割り込みラインを無効化する。
7425   lapicw(LINT0, MASKED);
7426   lapicw(LINT1, MASKED);
7427
7428   // 性能モニタリングカウンタオーバーフロー割り込みエントリが提供
7429   // されているマシンで、当該割り込みを無効化する。
7430   if(((lapic[VER]>>16) & 0xFF) >= 4)
7431     lapicw(PCINT, MASKED);
7432
7433   // エラー割り込みをIRQ_ERRORにマッピングする。
7434   lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
7435
7436   // エラーステータスレジスタをクリアする（連続書き込みが必要）
7437   lapicw(ESR, 0);
7438   lapicw(ESR, 0);
7439
7440   // 未処理のすべての割り込みを確認する。
7441   lapicw(EOI, 0);
7442
7443   // Init/レベルトリガ/レベルデアサートを送信して、アービトレーションIDを同期する。
7444   lapicw(ICRHI, 0);
7445   lapicw(ICRLO, BCAST | INIT | LEVEL);
7446   while(lapic[ICRLO] & DELIVS)
7447     ;
7448
7449
```



```
7450   // APIC上（プロセッサ上ではない）での割り込みを有効化する。
7451   lapicw(TPR, 0);
7452 }
7453
7454 int
7455 lapicid(void)
7456 {
7457   if (!lapic)
7458     return 0;
7459   return lapic[ID] >> 24;
7460 }
7461
7462 // 割り込みを確認する。
7463 void
7464 lapiceoi(void)
7465 {
7466   if(lapic)
7467     lapicw(EOI, 0);
7468 }
7469
7470 // 指定されたマイクロ秒数だけスピンする。
7471 // 実際のハードウェアではこれを動的に調整したいだろう。
7472 void
7473 microdelay(int us)
7474 {
7475 }
7476
7477 #define CMOS_PORT    0x70
7478 #define CMOS_RETURN  0x71
7479
7480 // addrにあるエントリコードを追加プロセッサで実行開始する
7481 // マルチプロセッサ仕様の付録Bを参照
7482 void
7483 lapicstartap(uchar apicid, uint addr)
7484 {
7485   int i;
7486   ushort *wrv;
7487
7488   // "BSPは、[汎用スタートアップアルゴリズム]の前に、CMOSシャットダウン
7489   // コードを0AHに、warmリセットベクタ（40:67をベースとするDWORD）を
7490   // APスタートアップコードを指すように初期化しなければならない。
7491   outb(CMOS_PORT, 0xF);  // オフセット0xFはシャットダウンコード
7492   outb(CMOS_PORT+1, 0x0A);
7493   wrv = (ushort*)P2V((0x40<<4 | 0x67));  // Warmリセットベクタ
7494   wrv[0] = 0;
7495   wrv[1] = addr >> 4;
7496
7497
7498
7499
```

```
7500   // "汎用スタートアップアルゴリズム"
7501   // INIT (レベルトリガ)割り込みを送信して他のCPUをリセットする
7502   lapicw(ICRHI, apicid<<24);
7503   lapicw(ICRLO, INIT ¦ LEVEL ¦ ASSERT);
7504   microdelay(200);
7505   lapicw(ICRLO, INIT ¦ LEVEL);
7506   microdelay(100);    // 10msでなければならないが、Bochsは遅すぎる!
7507
7508   // スタートアップIPIを( 2回!)送信してコードに入る。
7509   // 一般的なハードウェアは、INITによる停止中にある場合、
7510   // STARTUPを1回しか受け付けないと思われる。そのため、2回目は
7511   // 無視されるはずであるが、これがIntel公式のアルゴリズムである。
7512   // Bochsは2回めの送信でエラーコードをはく。Bochsにとっては最悪だ。
7513   for(i = 0; i < 2; i++){
7514     lapicw(ICRHI, apicid<<24);
7515     lapicw(ICRLO, STARTUP ¦ (addr>>12));
7516     microdelay(200);
7517   }
7518 }
7519
7520 #define CMOS_STATA   0x0a
7521 #define CMOS_STATB   0x0b
7522 #define CMOS_UIP    (1 << 7)        // 進行中にRTCを更新
7523
7524 #define SECS    0x00
7525 #define MINS    0x02
7526 #define HOURS   0x04
7527 #define DAY     0x07
7528 #define MONTH   0x08
7529 #define YEAR    0x09
7530
7531 static uint cmos_read(uint reg)
7532 {
7533   outb(CMOS_PORT,  reg);
7534   microdelay(200);
7535
7536   return inb(CMOS_RETURN);
7537 }
7538
7539 static void fill_rtcdate(struct rtcdate *r)
7540 {
7541   r->second = cmos_read(SECS);
7542   r->minute = cmos_read(MINS);
7543   r->hour   = cmos_read(HOURS);
7544   r->day    = cmos_read(DAY);
7545   r->month  = cmos_read(MONTH);
7546   r->year   = cmos_read(YEAR);
7547 }
7548
7549
```



```
7550 // qemuは24時GWTを使用し、その値はBCDエンコードされているようだ。
7551 void cmostime(struct rtcdate *r)
7552 {
7553   struct rtcdate t1, t2;
7554   int sb, bcd;
7555
7556   sb = cmos_read(CMOS_STATB);
7557
7558   bcd = (sb & (1 << 2)) == 0;
7559
7560   // 読み込み中にCMOSが時間を変更しないようにする
7561   for(;;) {
7562     fill_rtcdate(&t1);
7563     if(cmos_read(CMOS_STATA) & CMOS_UIP)
7564         continue;
7565     fill_rtcdate(&t2);
7566     if(memcmp(&t1, &t2, sizeof(t1)) == 0)
7567       break;
7568   }
7569
7570   // 変換する
7571   if(bcd) {
7572 #define    CONV(x)     (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
7573     CONV(second);
7574     CONV(minute);
7575     CONV(hour  );
7576     CONV(day   );
7577     CONV(month );
7578     CONV(year  );
7579 #undef     CONV
7580   }
7581
7582   *r = t1;
7583   r->year += 2000;
7584 }
7585
7586
7587
7588
7589
7590
7591
7592
7593
7594
7595
7596
7597
7598
7599
```

```
7600 // I/O APICはSMPシステムのハードウェア割り込みを管理する。
7601 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
7602 // picirq.cも参照のこと。
7603
7604 #include "types.h"
7605 #include "defs.h"
7606 #include "traps.h"
7607
7608 #define IOAPIC  0xFEC00000   // IO APICのデフォルト物理アドレス
7609
7610 #define REG_ID     0x00  // レジスタインデックス: ID
7611 #define REG_VER    0x01  // レジスタインデックス: バージョン
7612 #define REG_TABLE  0x10  // リダイレクションテーブルベース
7613
7614 // リダイレクションテーブルはREG_TABLEから始まり、
7615 // 2つのレジスタを使って、各割り込みを構成する。
7616 // 最初の（低位）レジスタは構成ビットを持つ。
7617 // 2番めの（高位）レジスタは、どのCPUがその割り込みに対応できるかを
7618 // ╕すビットマスクを持つ。
7619 #define INT_DISABLED   0x00010000  // 割り込みは禁止
7620 #define INT_LEVEL      0x00008000  // レベルトリガ（vs エッジトリガ）
7621 #define INT_ACTIVELOW  0x00002000  // アクティブロー（vs アクティブハイ）
7622 #define INT_LOGICAL    0x00000800  // 宛先はCPU id（vs APIC ID）
7623
7624 volatile struct ioapic *ioapic;
7625
7626 // IO APIC MMIO 構造体: regに書き込み、次にdataの読み込み/書き込みを行う。
7627 struct ioapic {
7628   uint reg;
7629   uint pad[3];
7630   uint data;
7631 };
7632
7633 static uint
7634 ioapicread(int reg)
7635 {
7636   ioapic->reg = reg;
7637   return ioapic->data;
7638 }
7639
7640 static void
7641 ioapicwrite(int reg, uint data)
7642 {
7643   ioapic->reg = reg;
7644   ioapic->data = data;
7645 }
7646
7647
7648
7649
```



```
7650 void
7651 ioapicinit(void)
7652 {
7653   int i, id, maxintr;
7654
7655   ioapic = (volatile struct ioapic*)IOAPIC;
7656   maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
7657   id = ioapicread(REG_ID) >> 24;
7658   if(id != ioapicid)
7659     cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
7660
7661   // すべての割り込みの設定を、エッジトリガ、アクティブハイ、割り込み無効、
7662   // どのCPUにも転送しない、とする。
7663   for(i = 0; i <= maxintr; i++){
7664     ioapicwrite(REG_TABLE+2*i, INT_DISABLED ¦ (T_IRQ0 + i));
7665     ioapicwrite(REG_TABLE+2*i+1, 0);
7666   }
7667 }
7668
7669 void
7670 ioapicenable(int irq, int cpunum)
7671 {
7672   // 割り込みの設定を、エッジトリガ、アクティブハイ、割り込み有効、
7673   // 指定されたcpunumに転送する、とする。
7674   // cpunumはそのcpuのAPIC IDでもある。
7675   ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
7676   ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
7677 }
7678
7679
7680
7681
7682
7683
7684
7685
7686
7687
7688
7689
7690
7691
7692
7693
7694
7695
7696
7697
7698
7699
```

```
7700 // PCキーボードインターフェース関連の定数
7701
7702 #define KBSTATP         0x64    // キーボードコントローラステータスポート(I)
7703 #define KBS_DIB         0x01    // バッファのキーボードデータ
7704 #define KBDATAP         0x60    // キーボードデータポート(I)
7705
7706 #define NO              0
7707
7708 #define SHIFT           (1<<0)
7709 #define CTL             (1<<1)
7710 #define ALT             (1<<2)
7711
7712 #define CAPSLOCK        (1<<3)
7713 #define NUMLOCK         (1<<4)
7714 #define SCROLLLOCK      (1<<5)
7715
7716 #define E0ESC           (1<<6)
7717
7718 // 特別なキーコード
7719 #define KEY_HOME        0xE0
7720 #define KEY_END         0xE1
7721 #define KEY_UP          0xE2
7722 #define KEY_DN          0xE3
7723 #define KEY_LF          0xE4
7724 #define KEY_RT          0xE5
7725 #define KEY_PGUP        0xE6
7726 #define KEY_PGDN        0xE7
7727 #define KEY_INS         0xE8
7728 #define KEY_DEL         0xE9
7729
7730 // C('A') == Control-A 制御コードC0集合の値 ^@ = 0x00 ^A = 0x01 ^B = 0x02
7731 #define C(x) (x - '@')
7732 // 以下はスキャンコードセット1のコード（メイクコード: 押した時のコード）
7733 static uchar shiftcode[256] =
7734 {
7735   [0x1D] CTL,
7736   [0x2A] SHIFT,
7737   [0x36] SHIFT,
7738   [0x38] ALT,
7739   [0x9D] CTL,
7740   [0xB8] ALT
7741 };
7742
7743 static uchar togglecode[256] =
7744 {
7745   [0x3A] CAPSLOCK,
7746   [0x45] NUMLOCK,
7747   [0x46] SCROLLLOCK
7748 };
7749
```



```
7750 static uchar normalmap[256] =
7751 {
7752   NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
7753   '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
7754   'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
7755   'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
7756   'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
7757   '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
7758   'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
7759   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
7760   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
7761   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
7762   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
7763   [0x9C] '\n',      // KP_Enter                    // ここからは独自指定
7764   [0xB5] '/',       // KP_Div
7765   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7766   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7767   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7768   [0x97] KEY_HOME,  [0xCF] KEY_END,
7769   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7770 };
7771
7772 static uchar shiftmap[256] =
7773 {
7774   NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
7775   '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
7776   'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
7777   'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
7778   'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
7779   '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
7780   'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
7781   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
7782   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
7783   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
7784   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
7785   [0x9C] '\n',      // KP_Enter
7786   [0xB5] '/',       // KP_Div
7787   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7788   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7789   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7790   [0x97] KEY_HOME,  [0xCF] KEY_END,
7791   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7792 };
7793
7794
7795
7796
7797
7798
7799
```

```
7800 static uchar ctlmap[256] =
7801 {
7802   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7803   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7804   C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
7805   C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
7806   C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
7807   NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
7808   C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
7809   [0x9C] '\r',      // KP_Enter
7810   [0xB5] C('/'),    // KP_Div
7811   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7812   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7813   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7814   [0x97] KEY_HOME,  [0xCF] KEY_END,
7815   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7816 };
7817
7818
7819
7820
7821
7822
7823
7824
7825
7826
7827
7828
7829
7830
7831
7832
7833
7834
7835
7836
7837
7838
7839
7840
7841
7842
7843
7844
7845
7846
7847
7848
7849
```



```
7850 #include "types.h"
7851 #include "x86.h"
7852 #include "defs.h"
7853 #include "kbd.h"
7854
7855 int
7856 kbdgetc(void)
7857 {
7858   static uint shift;
7859   static uchar *charcode[4] = {
7860     normalmap, shiftmap, ctlmap, ctlmap
7861   };
7862   uint st, data, c;
7863
7864   st = inb(KBSTATP);
7865   if((st & KBS_DIB) == 0)  // キーデータなし
7866     return -1;
7867   data = inb(KBDATAP);     // キーデータを読み取る
7868
7869   if(data == 0xE0){
7870     shift ¦= E0ESC;
7871     return 0;
7872   } else if(data & 0x80){
7873     // キーが離された
7874     data = (shift & E0ESC ? data : data & 0x7F);  // ブレイクコードをメイクコードに変換
7875     shift &= □(shiftcode[data] ¦ E0ESC);          // シフトコード解除
7876     return 0;
7877   } else if(shift & E0ESC){
7878     // 直前の文字が E0 escapeだった; 0x80と共に押された
7879     data ¦= 0x80;                                // ブレイクコードに変換
7880     shift &= □E0ESC;                             // シフトコード解除
7881   }
7882
7883   shift ¦= shiftcode[data];
7884   shift ^= togglecode[data];
7885   c = charcode[shift & (CTL ¦ SHIFT)][data];
7886   if(shift & CAPSLOCK){
7887     if('a' <= c && c <= 'z')
7888       c += 'A' - 'a';
7889     else if('A' <= c && c <= 'Z')
7890       c += 'a' - 'A';
7891   }
7892   return c;
7893 }
7894
7895 void
7896 kbdintr(void)
7897 {
7898   consoleintr(kbdgetc);
7899 }
```

```
7900 // コンソール入出力。
7901 // 入力はキーボードまたはシリアルポートから。
7902 // 出力はスクリーンとシリアルポートに書き込まれる。
7903
7904 #include "types.h"
7905 #include "defs.h"
7906 #include "param.h"
7907 #include "traps.h"
7908 #include "spinlock.h"
7909 #include "sleeplock.h"
7910 #include "fs.h"
7911 #include "file.h"
7912 #include "memlayout.h"
7913 #include "mmu.h"
7914 #include "proc.h"
7915 #include "x86.h"
7916
7917 static void consputc(int);
7918
7919 static int panicked = 0;
7920
7921 static struct {
7922   struct spinlock lock;
7923   int locking;
7924 } cons;
7925
7926 static void
7927 printint(int xx, int base, int sign)
7928 {
7929   static char digits[] = "0123456789abcdef";
7930   char buf[16];
7931   int i;
7932   uint x;
7933
7934   if(sign && (sign = xx < 0))
7935     x = -xx;
7936   else
7937     x = xx;
7938                                   // -1234
7939   i = 0;                          // buf = [4, 3, 2, 1, -]
7940   do{
7941     buf[i++] = digits[x % base];
7942   }while((x /= base) != 0);
7943
7944   if(sign)
7945     buf[i++] = '-';               // buf[4] = '-', i = 5
7946
7947   while(--i >= 0)
7948     consputc(buf[i]);             // putc => -1234
7949 }
```

```
8000 // コンソールにプリントする。%d, %x, %p, %sのみ理解する。
8001 void
8002 cprintf(char *fmt, ...)
8003 {
8004   int i, c, locking;
8005   uint *argp;
8006   char *s;
8007
8008   locking = cons.locking;
8009   if(locking)
8010     acquire(&cons.lock);
8011
8012   if (fmt == 0)
8013     panic("null fmt");
8014
8015   argp = (uint*)(void*)(&fmt + 1);
8016   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
8017     if(c != '%'){
8018       consputc(c);
8019       continue;
8020     }
8021     c = fmt[++i] & 0xff;
8022     if(c == 0)
8023       break;
8024     switch(c){
8025     case 'd':
8026       printint(*argp++, 10, 1);
8027       break;
8028     case 'x':
8029     case 'p':
8030       printint(*argp++, 16, 0);
8031       break;
8032     case 's':
8033       if((s = (char*)*argp++) == 0)
8034         s = "(null)";
8035       for(; *s; s++)
8036         consputc(*s);
8037       break;
8038     case '%':
8039       consputc('%');
8040       break;
8041     default:
8042       // 注意を引くために、未知の%シーケンスをプリントする。
8043       consputc('%');
8044       consputc(c);
8045       break;
8046     }
8047   }
8048
8049
```



```
8050   if(locking)
8051     release(&cons.lock);
8052 }
8053
8054 void
8055 panic(char *s)
8056 {
8057   int i;
8058   uint pcs[10];
8059
8060   cli();
8061   cons.locking = 0;
8062   // mycpu()からpanicを呼べるように、lapiccpunumを使用する。
8063   cprintf("lapicid %d: panic: ", lapicid());
8064   cprintf(s);
8065   cprintf("\n");
8066   getcallerpcs(&s, pcs);
8067   for(i=0; i<10; i++)
8068     cprintf(" %p", pcs[i]);
8069   panicked = 1; // 他のCPUをフリーズさせる。
8070   for(;;)
8071     ;
8072 }
8073
8074
8075
8076
8077
8078
8079
8080
8081
8082
8083
8084
8085
8086
8087
8088
8089
8090
8091
8092
8093
8094
8095
8096
8097
8098
8099
```

```
#define BACKSPACE 0x100
#define CRTPORT 0x3d4
static ushort *crt = (ushort*)P2V(0xb8000);  // CGAメモリ

static void
cgaputc(int c)
{
  int pos;

  // カーソル位置: col + 80*row.
  outb(CRTPORT, 14);
  pos = inb(CRTPORT+1) << 8;
  outb(CRTPORT, 15);
  pos |= inb(CRTPORT+1);

  if(c == '\n')
    pos += 80 - pos%80;
  else if(c == BACKSPACE){
    if(pos > 0) --pos;
  } else
    crt[pos++] = (c&0xff) | 0x0700;  // 白地に黒

  if(pos < 0 || pos > 25*80)
    panic("pos under/overflow");

  if((pos/80) >= 24){  // スクロールアップ。
    memmove(crt, crt+80, sizeof(crt[0])*23*80);
    pos -= 80;
    memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
  }

  outb(CRTPORT, 14);
  outb(CRTPORT+1, pos>>8);
  outb(CRTPORT, 15);
  outb(CRTPORT+1, pos);
  crt[pos] = ' ' | 0x0700;
}
```



```
void
consputc(int c)
{
  if(panicked){
    cli();
    for(;;)
      ;
  }

  if(c == BACKSPACE){
    uartputc('\b'); uartputc(' '); uartputc('\b');
  } else
    uartputc(c);
  cgaputc(c);
}

#define INPUT_BUF 128
struct {
  char buf[INPUT_BUF];
  uint r;  // 読み込みインデックス
  uint w;  // 書き込みインデックス
  uint e;  // 編集インデックス
} input;

#define C(x)  ((x)-'@')  // Control-x

void
consoleintr(int (*getc)(void))
{
  int c, doprocdump = 0;

  acquire(&cons.lock);
  while((c = getc()) >= 0){
    switch(c){
    case C('P'):  // プロセス一覧を表示する。
      // procdump() はcons.lockを間接的にロックする; 後で呼び出す。
      doprocdump = 1;
      break;
    case C('U'):  // １行削除。
      while(input.e != input.w &&
            input.buf[(input.e-1) % INPUT_BUF] != '\n'){
        input.e--;
        consputc(BACKSPACE);
      }
      break;
    case C('H'): case '\x7f':  // バックスペース
      if(input.e != input.w){
        input.e--;
        consputc(BACKSPACE);
      }
```

```
8200       break;
8201     default:
8202       if(c != 0 && input.e-input.r < INPUT_BUF){
8203         c = (c == '\r') ? '\n' : c;
8204         input.buf[input.e++ % INPUT_BUF] = c;
8205         consputc(c);
8206         if(c == '\n' ¦¦ c == C('D') ¦¦ input.e == input.r+INPUT_BUF){
8207           input.w = input.e;
8208           wakeup(&input.r);
8209         }
8210       }
8211       break;
8212     }
8213   }
8214   release(&cons.lock);
8215   if(doprocdump) {
8216     procdump();  // ここでprocdump()を呼び出す。cons.lockが保持される。
8217   }
8218 }
8219
8220 int
8221 consoleread(struct inode *ip, char *dst, int n)
8222 {
8223   uint target;
8224   int c;
8225
8226   iunlock(ip);
8227   target = n;
8228   acquire(&cons.lock);
8229   while(n > 0){
8230     while(input.r == input.w){
8231       if(myproc()->killed){
8232         release(&cons.lock);
8233         ilock(ip);
8234         return -1;
8235       }
8236       sleep(&input.r, &cons.lock);
8237     }
8238     c = input.buf[input.r++ % INPUT_BUF];
8239     if(c == C('D')){  // EOF
8240       if(n < target){
8241         // 次回、呼び出し側が0バイトの結果を得られるように、
8242         // ^D を保存する。
8243         input.r--;
8244       }
8245       break;
8246     }
8247     *dst++ = c;
8248     --n;
8249     if(c == '\n')
```



```
8250       break;
8251   }
8252   release(&cons.lock);
8253   ilock(ip);
8254
8255   return target - n;
8256 }
8257
8258 int
8259 consolewrite(struct inode *ip, char *buf, int n)
8260 {
8261   int i;
8262
8263   iunlock(ip);
8264   acquire(&cons.lock);
8265   for(i = 0; i < n; i++)
8266     consputc(buf[i] & 0xff);
8267   release(&cons.lock);
8268   ilock(ip);
8269
8270   return n;
8271 }
8272
8273 void
8274 consoleinit(void)
8275 {
8276   initlock(&cons.lock, "console");
8277
8278   devsw[CONSOLE].write = consolewrite;
8279   devsw[CONSOLE].read = consoleread;
8280   cons.locking = 1;
8281
8282   ioapicenable(IRQ_KBD, 0);
8283 }
8284
8285
8286
8287
8288
8289
8290
8291
8292
8293
8294
8295
8296
8297
8298
8299
```

```
8300 // Intel 8250シリアルポート(UART)。
8301
8302 #include "types.h"
8303 #include "defs.h"
8304 #include "param.h"
8305 #include "traps.h"
8306 #include "spinlock.h"
8307 #include "sleeplock.h"
8308 #include "fs.h"
8309 #include "file.h"
8310 #include "mmu.h"
8311 #include "proc.h"
8312 #include "x86.h"
8313
8314 #define COM1    0x3f8
8315
8316 static int uart;    // uartがあるか?
8317
8318 void
8319 uartinit(void)
8320 {
8321   char *p;
8322
8323   // FIFOを止める
8324   outb(COM1+2, 0);
8325
8326   // 9600 ボー、8 データビット、1 ストップビット、パリティオフ。
8327   outb(COM1+3, 0x80);    // divisorのロックを外す
8328   outb(COM1+0, 115200/9600);
8329   outb(COM1+1, 0);
8330   outb(COM1+3, 0x03);    // divsorをロックする、8 データビット。
8331   outb(COM1+4, 0);
8332   outb(COM1+1, 0x01);    // 受信割り込みを有効にする。
8333
8334   // ステータスが0xFFの場合、シリアルポートがない。
8335   if(inb(COM1+5) == 0xFF)
8336     return;
8337   uart = 1;
8338
8339   // 事前の割り込み条件を確認する。
8340   // 割り込みを有効にする。
8341   inb(COM1+2);
8342   inb(COM1+0);
8343   ioapicenable(IRQ_COM1, 0);
8344
8345   // ここにいることを通知する。
8346   for(p="xv6...\n"; *p; p++)
8347     uartputc(*p);
8348 }
8349
```



```
8350 void
8351 uartputc(int c)
8352 {
8353   int i;
8354
8355   if(!uart)
8356     return;
8357   for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
8358     microdelay(10);
8359   outb(COM1+0, c);
8360 }
8361
8362 static int
8363 uartgetc(void)
8364 {
8365   if(!uart)
8366     return -1;
8367   if(!(inb(COM1+5) & 0x01))
8368     return -1;
8369   return inb(COM1+0);
8370 }
8371
8372 void
8373 uartintr(void)
8374 {
8375   consoleintr(uartgetc);
8376 }
8377
8378
8379
8380
8381
8382
8383
8384
8385
8386
8387
8388
8389
8390
8391
8392
8393
8394
8395
8396
8397
8398
8399
```

```
# 最初のプロセスは /init をexec。
# このコードはユーザ空間で実行する。

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


# exec(init, argv)
.globl start
start:
  pushl $argv
  pushl $init
  pushl $0  // 呼び出し側のpcはここになる
  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
```



```
#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)
```

```
8500 // init: 最初のユーザレベルのプログラム
8501
8502 #include "types.h"
8503 #include "stat.h"
8504 #include "user.h"
8505 #include "fcntl.h"
8506
8507 char *argv[] = { "sh", 0 };
8508
8509 int
8510 main(void)
8511 {
8512   int pid, wpid;
8513
8514   if(open("console", O_RDWR) < 0){
8515     mknod("console", 1, 1);
8516     open("console", O_RDWR);
8517   }
8518   dup(0);  // stdout
8519   dup(0);  // stderr
8520
8521   for(;;){
8522     printf(1, "init: starting sh\n");
8523     pid = fork();
8524     if(pid < 0){
8525       printf(1, "init: fork failed\n");
8526       exit();
8527     }
8528     if(pid == 0){
8529       exec("sh", argv);
8530       printf(1, "init: exec sh failed\n");
8531       exit();
8532     }
8533     while((wpid=wait()) >= 0 && wpid != pid)
8534       printf(1, "zombie!\n");
8535   }
8536 }
8537
8538
8539
8540
8541
8542
8543
8544
8545
8546
8547
8548
8549
```



```
8550 // シェル
8551
8552 #include "types.h"
8553 #include "user.h"
8554 #include "fcntl.h"
8555
8556 // パース後のコマンド表現
8557 #define EXEC  1
8558 #define REDIR 2
8559 #define PIPE  3
8560 #define LIST  4
8561 #define BACK  5
8562
8563 #define MAXARGS 10
8564
8565 struct cmd {
8566   int type;
8567 };
8568
8569 struct execcmd {
8570   int type;
8571   char *argv[MAXARGS];    // 引数文字列の先頭ポインタ
8572   char *eargv[MAXARGS];   // 引数文字列の末尾の次のポインタ
8573 };
8574
8575 struct redircmd {
8576   int type;
8577   struct cmd *cmd;
8578   char *file;
8579   char *efile;
8580   int mode;
8581   int fd;
8582 };
8583
8584 struct pipecmd {
8585   int type;
8586   struct cmd *left;
8587   struct cmd *right;
8588 };
8589
8590 struct listcmd {
8591   int type;
8592   struct cmd *left;
8593   struct cmd *right;
8594 };
8595
8596 struct backcmd {
8597   int type;
8598   struct cmd *cmd;
8599 };
```

```
8600 int fork1(void);  // エラー時にpanicを呼び出すfork
8601 void panic(char*);
8602 struct cmd *parsecmd(char*);
8603
8604 // cmdを実行する。復帰しない。
8605 void
8606 runcmd(struct cmd *cmd)
8607 {
8608   int p[2];
8609   struct backcmd *bcmd;
8610   struct execcmd *ecmd;
8611   struct listcmd *lcmd;
8612   struct pipecmd *pcmd;
8613   struct redircmd *rcmd;
8614
8615   if(cmd == 0)
8616     exit();
8617
8618   switch(cmd->type){
8619   default:
8620     panic("runcmd");
8621
8622   case EXEC:
8623     ecmd = (struct execcmd*)cmd;
8624     if(ecmd->argv[0] == 0)
8625       exit();
8626     exec(ecmd->argv[0], ecmd->argv);
8627     printf(2, "exec %s failed\n", ecmd->argv[0]);
8628     break;
8629
8630   case REDIR:
8631     rcmd = (struct redircmd*)cmd;
8632     close(rcmd->fd);
8633     if(open(rcmd->file, rcmd->mode) < 0){
8634       printf(2, "open %s failed\n", rcmd->file);
8635       exit();
8636     }
8637     runcmd(rcmd->cmd);
8638     break;
8639
8640   case LIST:
8641     lcmd = (struct listcmd*)cmd;
8642     if(fork1() == 0)
8643       runcmd(lcmd->left);
8644     wait();
8645     runcmd(lcmd->right);
8646     break;
8647
8648
8649
```



```
8650   case PIPE:
8651     pcmd = (struct pipecmd*)cmd;
8652     if(pipe(p) < 0)
8653       panic("pipe");
8654     if(fork1() == 0){
8655       close(1);
8656       dup(p[1]);
8657       close(p[0]);
8658       close(p[1]);
8659       runcmd(pcmd->left);
8660     }
8661     if(fork1() == 0){
8662       close(0);
8663       dup(p[0]);
8664       close(p[0]);
8665       close(p[1]);
8666       runcmd(pcmd->right);
8667     }
8668     close(p[0]);
8669     close(p[1]);
8670     wait();
8671     wait();
8672     break;
8673
8674   case BACK:
8675     bcmd = (struct backcmd*)cmd;
8676     if(fork1() == 0)
8677       runcmd(bcmd->cmd);
8678     break;
8679   }
8680   exit();
8681 }
8682
8683 int
8684 getcmd(char *buf, int nbuf)
8685 {
8686   printf(2, "$ ");
8687   memset(buf, 0, nbuf);
8688   gets(buf, nbuf);
8689   if(buf[0] == 0) // EOF
8690     return -1;
8691   return 0;
8692 }
8693
8694
8695
8696
8697
8698
8699
```

```
8700 int
8701 main(void)
8702 {
8703   static char buf[100];
8704   int fd;
8705
8706   // 3つのファイル記述子がオープンされていることを保証する。
8707   while((fd = open("console", O_RDWR)) >= 0){
8708     if(fd >= 3){
8709       close(fd);
8710       break;
8711     }
8712   }
8713
8714   // 入力されたコマンドを読み込んで実行する。
8715   while(getcmd(buf, sizeof(buf)) >= 0){
8716     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
8717       // chdir は子プロセスではなく、親プロセスから呼ばれなければならない。
8718       buf[strlen(buf)-1] = 0;  // \nを捨てる
8719       if(chdir(buf+3) < 0)
8720         printf(2, "cannot cd %s\n", buf+3);
8721       continue;
8722     }
8723     if(fork1() == 0)
8724       runcmd(parsecmd(buf));
8725     wait();
8726   }
8727   exit();
8728 }
8729
8730 void
8731 panic(char *s)
8732 {
8733   printf(2, "%s\n", s);
8734   exit();
8735 }
8736
8737 int
8738 fork1(void)
8739 {
8740   int pid;
8741
8742   pid = fork();
8743   if(pid == -1)
8744     panic("fork");
8745   return pid;
8746 }
8747
8748
8749
```



```
8750 // コンストラクタ
8751
8752 struct cmd*
8753 execcmd(void)
8754 {
8755   struct execcmd *cmd;
8756
8757   cmd = malloc(sizeof(*cmd));
8758   memset(cmd, 0, sizeof(*cmd));
8759   cmd->type = EXEC;
8760   return (struct cmd*)cmd;
8761 }
8762
8763 struct cmd*
8764 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
8765 {
8766   struct redircmd *cmd;
8767
8768   cmd = malloc(sizeof(*cmd));
8769   memset(cmd, 0, sizeof(*cmd));
8770   cmd->type = REDIR;
8771   cmd->cmd = subcmd;
8772   cmd->file = file;
8773   cmd->efile = efile;
8774   cmd->mode = mode;
8775   cmd->fd = fd;
8776   return (struct cmd*)cmd;
8777 }
8778
8779 struct cmd*
8780 pipecmd(struct cmd *left, struct cmd *right)
8781 {
8782   struct pipecmd *cmd;
8783
8784   cmd = malloc(sizeof(*cmd));
8785   memset(cmd, 0, sizeof(*cmd));
8786   cmd->type = PIPE;
8787   cmd->left = left;
8788   cmd->right = right;
8789   return (struct cmd*)cmd;
8790 }
8791
8792
8793
8794
8795
8796
8797
8798
8799
```

```
8800 struct cmd*
8801 listcmd(struct cmd *left, struct cmd *right)
8802 {
8803   struct listcmd *cmd;
8804
8805   cmd = malloc(sizeof(*cmd));
8806   memset(cmd, 0, sizeof(*cmd));
8807   cmd->type = LIST;
8808   cmd->left = left;
8809   cmd->right = right;
8810   return (struct cmd*)cmd;
8811 }
8812
8813 struct cmd*
8814 backcmd(struct cmd *subcmd)
8815 {
8816   struct backcmd *cmd;
8817
8818   cmd = malloc(sizeof(*cmd));
8819   memset(cmd, 0, sizeof(*cmd));
8820   cmd->type = BACK;
8821   cmd->cmd = subcmd;
8822   return (struct cmd*)cmd;
8823 }
8824
8825
8826
8827
8828
8829
8830
8831
8832
8833
8834
8835
8836
8837
8838
8839
8840
8841
8842
8843
8844
8845
8846
8847
8848
8849
```



```
8850 // パース処理
8851
8852 char whitespace[] = " \t\r\n\v";
8853 char symbols[] = "<|>&;()";
8854
8855 int
8856 gettoken(char **ps, char *es, char **q, char **eq)
8857 {
8858   char *s;
8859   int ret;
8860
8861   s = *ps;
8862   while(s < es && strchr(whitespace, *s))  // ホワイトスペースを読み飛ばす
8863     s++;
8864   if(q)
8865     *q = s;
8866   ret = *s;
8867   switch(*s){
8868   case 0:
8869     break;
8870   case '|':
8871   case '(':
8872   case ')':
8873   case ';':
8874   case '&':
8875   case '<':
8876     s++;
8877     break;
8878   case '>':
8879     s++;
8880     if(*s == '>'){   //  >> : 追加モード
8881       ret = '+';
8882       s++;
8883     }
8884     break;
8885   default:
8886     ret = 'a';       // ホワイトスペースでもシンボルでもない文字を読み飛ばす
8887     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
8888       s++;
8889     break;
8890   }
8891   if(eq)
8892     *eq = s;
8893
8894   while(s < es && strchr(whitespace, *s))  // ホワイトスペースを読み飛ばす
8895     s++;
8896   *ps = s;
8897   return ret;  // シンボルは自身 ( ">>"は'+' ) , 文字は"a"を返す。*psは次の有効文字を指す
8898 }
8899
```

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

  s = *ps;
  while(s < es && strchr(whitespace, *s))  // ホワイトスペースを読み飛ばす
    s++;
  *ps = s;
  return *s && strchr(toks, *s);           // 先頭文字がtoksか？
}

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;             // s + strlen(s): 処理の終了判定に使用
  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;
                            // PIPE, BACK, LISTを処理する
  cmd = parsepipe(ps, es);  // コマンドをパースして
  while(peek(ps, es, "&")){ // 次の文字が"&"ならバックグラウンド処理
    gettoken(ps, es, 0, 0); // ポインタを進めて
    cmd = backcmd(cmd);     // backcmdを作成
  }                         // "&&"コマンドの処理がこれではおかしいが想定外? だとしたらなぜwh
  if(peek(ps, es, ";")){    // 次の文字が";"ならコマンドの連速実行(listcmd)
    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);   // パイプの左側のコマンドを取得
  if(peek(ps, es, "¦")){     // 次の文字が"¦"の場合はパイプライン
    gettoken(ps, es, 0, 0);  // 返り値は"¦"だが無視して、ポインタを進める
    cmd = pipecmd(cmd, parsepipe(ps, es));  // パイプの右側をパースしてpipecmdを作成
  }
  return cmd;
}

struct cmd*
parseredirs(struct cmd *cmd, char **ps, char *es)
{
  int tok;
  char *q, *eq;
  // *psが<>の場合のみ処理、それ以外はcmdをそのまま返す
  while(peek(ps, es, "<>")){
    tok = gettoken(ps, es, 0, 0);          // tok = [<, >、+]
    if(gettoken(ps, es, &q, &eq) != 'a')   // q = このtokenの先頭、eq = このtokenの最後
      panic("missing file for redirection");
    switch(tok){
    case '<':
      cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
      break;
    case '>':
      cmd = redircmd(cmd, q, eq, O_WRONLY¦O_CREATE, 1);
      break;
    case '+':  // >>             これでは追加にならないのでは。O_CREATEは不要?
      cmd = redircmd(cmd, q, eq, O_WRONLY¦O_CREATE, 1);
      break;
    }
  }
  return cmd;
}
```

```
9000 struct cmd*
9001 parseblock(char **ps, char *es)
9002 {
9003   struct cmd *cmd;
9004
9005   if(!peek(ps, es, "("))
9006     panic("parseblock");
9007   gettoken(ps, es, 0, 0);
9008   cmd = parseline(ps, es);         // ( EXEC¦PIPE¦LIST¦BACK )
9009   if(!peek(ps, es, ")"))
9010     panic("syntax - missing )");
9011   gettoken(ps, es, 0, 0);
9012   cmd = parseredirs(cmd, ps, es); // ( EXEC¦PIPE¦LIST¦BACK ) <>> ファイル
9013   return cmd;
9014 }
9015
9016 struct cmd*
9017 parseexec(char **ps, char *es)
9018 {
9019   char *q, *eq;
9020   int tok, argc;
9021   struct execcmd *cmd;
9022   struct cmd *ret;
9023
9024   if(peek(ps, es, "("))         // ホワイトスペースを覗いた先頭文字が"("
9025     return parseblock(ps, es);
9026
9027   ret = execcmd();
9028   cmd = (struct execcmd*)ret;
9029
9030   argc = 0;
9031   ret = parseredirs(ret, ps, es);
9032   while(!peek(ps, es, "¦)&;")){   // コマンド引数の取得
9033     if((tok=gettoken(ps, es, &q, &eq)) == 0)
9034       break;
9035     if(tok != 'a')
9036       panic("syntax");
9037     cmd->argv[argc] = q;
9038     cmd->eargv[argc] = eq;
9039     argc++;
9040     if(argc >= MAXARGS)
9041       panic("too many args");
9042     ret = parseredirs(ret, ps, es);
9043   }
9044   cmd->argv[argc] = 0;
9045   cmd->eargv[argc] = 0;
9046   return ret;
9047 }
9048
9049
```



```
9050 // 文字列末尾の次を表すポインタに0を代入して文字列をヌル終端
9051 struct cmd*
9052 nulterminate(struct cmd *cmd)
9053 {
9054   int i;
9055   struct backcmd *bcmd;
9056   struct execcmd *ecmd;
9057   struct listcmd *lcmd;
9058   struct pipecmd *pcmd;
9059   struct redircmd *rcmd;
9060
9061   if(cmd == 0)
9062     return 0;
9063
9064   switch(cmd->type){
9065   case EXEC:
9066     ecmd = (struct execcmd*)cmd;
9067     for(i=0; ecmd->argv[i]; i++)     // 設定された引数だけ処理
9068       *ecmd->eargv[i] = 0;           // 末尾+1にヌルを代入して、argv[i]をヌル終端
9069     break;
9070
9071   case REDIR:
9072     rcmd = (struct redircmd*)cmd;
9073     nulterminate(rcmd->cmd);
9074     *rcmd->efile = 0;                // rcmd->fileをヌル終端
9075     break;
9076
9077   case PIPE:
9078     pcmd = (struct pipecmd*)cmd;
9079     nulterminate(pcmd->left);
9080     nulterminate(pcmd->right);
9081     break;
9082
9083   case LIST:
9084     lcmd = (struct listcmd*)cmd;
9085     nulterminate(lcmd->left);
9086     nulterminate(lcmd->right);
9087     break;
9088
9089   case BACK:
9090     bcmd = (struct backcmd*)cmd;
9091     nulterminate(bcmd->cmd);
9092     break;
9093   }
9094   return cmd;
9095 }
9096
9097
9098
9099
```

```
9100 #include "asm.h"
9101 #include "memlayout.h"
9102 #include "mmu.h"
9103
9104 # 1番目のCPUを起動する: 32ビットプロテクトモードに切り替えて
9105 # Cの関数にジャンプする。
9106 # BIOSはこのコードをHDの第1セクタから物理アドレス 0x7c00のメモリに
9107 # ロードし、%cs=0, %ip=7c00 のリアルモードで実行を開始する。
9108
9109 .code16                       # 16ビットモードでアセンブルする
9110 .globl start
9111 start:
9112   cli                         # BIOSが有効にした割り込みを禁止する。
9113
9114   # データセグメントレジスタ DS, ES, SS を0クリア。
9115   xorw    %ax,%ax             # %ax を0にセット
9116   movw    %ax,%ds             # -> データセグメント
9117   movw    %ax,%es             # -> 拡張セグメント
9118   movw    %ax,%ss             # -> スタックセグメント
9119
9120   # はじめて2MBのメモリを持ったPCで1MBを想定したソフトウェアを実行できるよう
9121   # 物理アドレスラインA20は0にセットされている。これを取り消す。
9122 seta20.1:
9123   inb     $0x64,%al               # busy状態が解消されるまで待機
9124   testb   $0x2,%al
9125   jnz     seta20.1
9126
9127   movb    $0xd1,%al               # 0xd1 -> port 0x64
9128   outb    %al,$0x64
9129
9130 seta20.2:
9131   inb     $0x64,%al               # busy状態が解消されるまで待機
9132   testb   $0x2,%al
9133   jnz     seta20.2
9134
9135   movb    $0xdf,%al               # 0xdf -> port 0x60
9136   outb    %al,$0x60
9137
9138   # リアルモードからプロテクトモードに切り替える。実効メモリマップが
9139   # 移行期間中に変更されないように、仮想アドレスをそのまま物理アドレスに
9140   # マッピングする起動用のGDTを使用する。
9141   lgdt    gdtdesc
9142   movl    %cr0, %eax
9143   orl     $CR0_PE, %eax
9144   movl    %eax, %cr0
9145
9146
9147
9148
9149
```



```
9150   # %csと%eipを再ロードするロングジャンプを使用することにより、32ビット
9151   # プロテクトモードへの移行を完了させる。セグメントディスクリプタは
9152   # 変換されることなく設定されるので、マッピングは恒等マッピングのままである。
9153   ljmp    $(SEG_KCODE<<3), $start32
9154
9155 .code32  # アセンブラにここからは32ビットコードを生成するように伝える。
9156 start32:
9157   # プロテクトモードのデータセグメントレジスタを設定する。
9158   movw    $(SEG_KDATA<<3), %ax    # 使用するデータセグメントセレクタ
9159   movw    %ax, %ds                # -> DS: データセグメント
9160   movw    %ax, %es                # -> ES: 拡張セグメント
9161   movw    %ax, %ss                # -> SS: スタックセグメント
9162   movw    $0, %ax                 # 未使用のセグメントは0クリア
9163   movw    %ax, %fs                # -> FS
9164   movw    %ax, %gs                # -> GS
9165
9166   # スタックポインタをセットして、Cの関数を呼び出す。
9167   movl    $start, %esp
9168   call    bootmain
9169
9170   # bootmainが復帰したら（しないはず）、Bochs下で実行している場合は
9171   # Bochsのブレークポイントを呼び出して、ループする。
9172   movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
9173   movw    %ax, %dx
9174   outw    %ax, %dx
9175   movw    $0x8ae0, %ax            # 0x8ae0 -> port 0x8a00
9176   outw    %ax, %dx
9177 spin:
9178   jmp     spin
9179
9180 # 起動用GDT
9181 .p2align 2                                # 4バイト境界に揃える
9182 gdt:
9183   SEG_NULLASM                             # ヌルセグメント
9184   SEG_ASM(STA_X¦STA_R, 0x0, 0xffffffff)   # コードセグメント
9185   SEG_ASM(STA_W, 0x0, 0xffffffff)         # データセグメント
9186
9187 gdtdesc:
9188   .word   (gdtdesc - gdt - 1)             # sizeof(gdt) - 1
9189   .long   gdt                             # gdtアドレス
9190
9191
9192
9193
9194
9195
9196
9197
9198
9199
```

```
9200 // ブートローダ。
9201 //
9202 // bootmain()を呼び出すbootasm.Sと共にブートブロックを形成する。
9203 // bootasm.Sは、プロセッサを32ビットプロテクトモードにした。
9204 // bootmain()は、ディスクのセクタ1からELFカーネルイメージをロードして、
9205 // カーネルのエントリルーチンにジャンプする。
9206
9207 #include "types.h"
9208 #include "elf.h"
9209 #include "x86.h"
9210 #include "memlayout.h"
9211
9212 #define SECTSIZE  512
9213
9214 void readseg(uchar*, uint, uint);
9215
9216 void
9217 bootmain(void)
9218 {
9219   struct elfhdr *elf;
9220   struct proghdr *ph, *eph;
9221   void (*entry)(void);
9222   uchar* pa;
9223
9224   elf = (struct elfhdr*)0x10000;  // 開始位置を設定
9225
9226   // ディスクから最初のページを読み込む
9227   readseg((uchar*)elf, 4096, 0);
9228
9229   // ELF実行ファイルか?
9230   if(elf->magic != ELF_MAGIC)
9231     return;  // bootasm.Sにエラー処理を任せる
9232
9233   // 各プログラムセグメントをロードする（phフラグは無視する）
9234   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
9235   eph = ph + elf->phnum;
9236   for(; ph < eph; ph++){
9237     pa = (uchar*)ph->paddr;
9238     readseg(pa, ph->filesz, ph->off);
9239     if(ph->memsz > ph->filesz)
9240       stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
9241   }
9242
9243   // ELFヘッダからエントリポイントを呼び出す。
9244   // 復帰しない!
9245   entry = (void(*)(void))(elf->entry);
9246   entry();
9247 }
9248
9249
```



```
9250 void
9251 waitdisk(void)
9252 {
9253   // ディスクの用意ができるのを待機する。
9254   while((inb(0x1F7) & 0xC0) != 0x40)
9255     ;
9256 }
9257
9258 // offsetからセクタを1つdstに読み込む。
9259 void
9260 readsect(void *dst, uint offset)
9261 {
9262   // コマンドを発行する。
9263   waitdisk();
9264   outb(0x1F2, 1);   // count = 1
9265   outb(0x1F3, offset);
9266   outb(0x1F4, offset >> 8);
9267   outb(0x1F5, offset >> 16);
9268   outb(0x1F6, (offset >> 24) | 0xE0);
9269   outb(0x1F7, 0x20);  // cmd 0x20 - セクタの読み込み
9270
9271   // データを読み込む。
9272   waitdisk();
9273   insl(0x1F0, dst, SECTSIZE/4);
9274 }
9275
9276 // カーネルの'offset'から'count'バイトを物理アドレス'pa'に読み込む。
9277 // 要求以上にコピーする場合がある。
9278 void
9279 readseg(uchar* pa, uint count, uint offset)
9280 {
9281   uchar* epa;
9282
9283   epa = pa + count;
9284
9285   // セクタ境界に切り捨てる。
9286   pa -= offset % SECTSIZE;
9287
9288   // バイトをセクタに変換する; カーネルはセクタ1から始まる。
9289   offset = (offset / SECTSIZE) + 1;
9290
9291   // これが遅すぎる場合は、一度にもっと多くのセクタを読み込めるかもしれない。
9292   // 要求以上にメモリに書き込む場合があるが、昇順でロードするので、
9293   // 問題はない。
9294   for(; pa < epa; pa += SECTSIZE, offset++)
9295     readsect(pa, offset);
9296 }
9297
9298
9299
```