kinit1(end,P2V(4*1024*1024))

内核的第一个调用，kinit()将第一块 4M 内存空间中除内核外的部分初始化，并添加到空闲内存链表中，交由内存管理器管理（allocator）。第一个参数 end 在文件 kernel.ld（63 行）中定义，其为一个地址，紧邻内核程序地址空间末尾，第二个参数 P2V（4*1024*1024）为 4M 物理地址的虚拟地址。接下来进入 kinit1 的定义：

```
void kinit1(void *vstart, void *vend)
```

第一步，调用 initlock（&kem.lock, "kmem"），初始化内核内存的锁，内核内存（kmem）结构体由 3 个成员组成，成员 lock 是内存锁，保证分配内存时的非竞争性，initlock 初始化这个锁，进入 initlock 的定义：

```
void initlock(struct spinlock *lk, char *name)
```

主要工作是给锁起个名字（这里是“kmem”）。回到 kinit1 的定义：

kmem 结构体的第二个成员 use_lock 置 0 时，表示不使用内存锁（在第一个用户进程启动前，都是不需要使用内存锁的），置 1 时，使用内存锁（每次分配内核内存前，必须上锁）。这里将 use_lock 置 0，暂不使用内存锁。

第二步，freerange（vstart，vend），将第一块 4M 内存空间中除内核外的部分初始化，进入 freerange 的定义：

```
void freerange(void *vstart, void* vend)
```

首先调用 PGROUNDUP 获取 vstart 地址之后的第一个页对齐的地址（空闲内存链表中必须都是页对齐的一页一页的地址），查看 PGROUNDUP 定义：

```
#define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
#define PGSIZE 4096
```

宏获取 sz 地址之后的第一个页对齐的地址。回到 freerange 的定义：

然后调用 kfree 将除内核外的第一个 4M 内存块一页一页的添加到空闲内存链表中。进入

kfree 的定义：

```
void kfree(char *v)
```

首先查看传入的地址是否合法（是否页对齐，是否在合法的地址空间内），接下来将这一页地址都填上垃圾数据（每个字节都赋予 1），然后获取 kmem 锁，进入 acquire 定义：

```
void acquire(struct spinlock *lk)
```

首先调用 pushcli 关中断，进入 pushcli 定义：

```
void pushcli(void)
```

首先调用 readeflags 获取 EFLAGS 寄存器数据，进入 readeflags 定义：

```
static inline uint readeflags(void)
```

直接调用 x86 指令（pushfl；popl）将 EFLAGS 寄存器的值取出并返回给调用程序。

回到 pushcli 定义，接着，调用 cli 关中断，进入 cli 定义：

```
static inline void cli(void)
```

直接调用 x86 指令 (cli) 将 EFLAGS 寄存器中的 IF 寄存器置零，达到关闭中断的目的。

回到 pushcli 定义，最后判断当前的关中断是否是第一层关中断，若是，则将 EFLAGS 寄存器中的 IF 寄存器的值存入 cpu->intena，便于恢复 IF 寄存器。值得注意的是，pushcli 函数以栈的形式来管理调用 cli 的次数，实际上只有第一次调用（栈为空时）cli 的会真正的有所动作（清 IF，存 IF），其余情况只是将调用 cli 的次数加 1（cpu->ncli++），这么做的原因是 xv6 中，加锁之前一定保证关中断，可很多动作需要不止一把锁，所以每加一把锁就记录一次（cpu->ncli++），只有所有锁都释放时，才会真正的恢复 IF 寄存器，释放锁的过程以后会详细说明。

回到 acquire 定义，接下来调用 holding 验证当前 cpu 是否已持有锁，进入 holding 定义：

```
int holding(struct spinlock *lock)
```

查看是否锁已被当前的 cpu 锁住。

回到 acquire 定义，若锁已被当前 cpu 持有，直接调用 panic 报错，内核崩溃，进入 panic 定义：

```
void panic(char *s)
```

关中断，打印出错信息及函数调用栈信息，最后所有 cpu 进入死循环，待补充。

回到 acquire 定义，接下来，不断调用 xchg 获取锁，进入 xchg 定义：

```
static inline uint xchg(volatile uint *addr, uint newval)
```

调用 x86 指令将 addr 地址上的值设置为 newval，并将旧的 addr 上的值返回，其中的 lock 指令保证 addr 地址上数据修改时的非竞争性。

回到 acquire 定义，然后，设置当前锁的 cpu 归属，最后，调用 getcallerpc 保存锁的本次使用记录，以供 debug 使用，进入 getcallerpc 定义：

```
void getcallerpcs(void *v, uint pcs[])
```

回溯函数的调用，将每一次函数调用的 eip 保存下来并返回（形成一个函数调用链，便于 debug），详见 Intel x86 Function-call Conventions。getcallerpc 定义结束。

回到 acquire 定义，acquire 定义结束。

回到 kfree 定义，然后，将这一页地址加到空闲页链表（kmem.freelist）的头部，最后，调用 release 释放 kmem 锁，进入 release 定义：

```
void release(struct spinlock *lk)
```

首先调用 holding 检测是否已持有锁，若是则 panic，接下来，将锁的除 name 以外的成员全部清零（调用 xchg 来清零 locked 成员），最后，调用 popcli 退到 cli 上一层，进入 popcli 定义：

```
void popcli(void)
```

首先调用 readflags 来检测中断是否已关闭，若否则 panic，接下来，将调用 cli 的次数减 1（--cpu->ncli，关于 ncli，可以结合 pushcli 的定义详细了解），并检测其合法性，不合法就 panic，最后查看当前锁是否是进程持有的唯一锁，若是则恢复 EFLAGS 寄存器中的 IF 标定位（调用 sti），进入 sti 的定义：

```
static inline void sti(void)
```

直接执行 x86 指令 sti 将 EFLAGS 寄存器中的 IF 标志位置 1，sti 定义结束。

回到 popcli 定义，popcli 定义结束。

回到 release 定义，release 定义结束。

回到 kfree 定义，kfree 定义结束。

回到 freerange 定义，freerange 定义结束。

回到 kinit1 定义，kinit 定义结束。

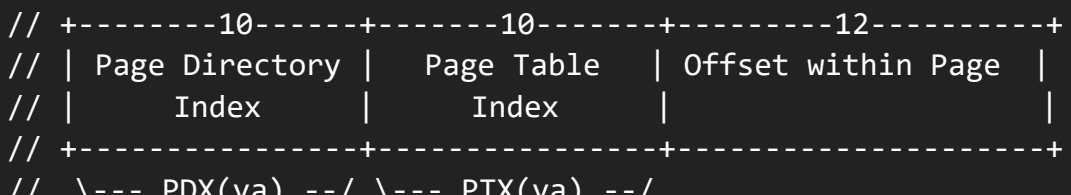

图 虚拟地址各个部分的含义

kvmalloc()

继续阅读之前，先浏览下上图“虚拟地址各个部分的含义”，接下来的内容会多次涉及到图中的内容。

接下来，内核调用 kvmalloc 来设置内核的页表，进入 kvmalloc 的定义：

```
void kvmalloc(void)
```

首先调用 setupkvm 新建一个内核页表目录，进入 setupkvm 的定义：

```
pde_t* setupkvm(void)
```

首先调用 kalloc 获取一页内存用来存放页表目录，接下来检测这一页内存地址的合法性，然后调用 mappages 将内核每一页物理地址到虚拟地址的映射存到各个页表中，进入

mappages 的定义：

```
static int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
```

首先调用 PGROUNDDOWN 来获得内核每个模块起始和结束虚拟地址所在的页的地址（页下对齐），接下来调用 walkpgdir 获取每一页虚拟内存对应的物理内存在页表中的存储地址，进入 walkpgdir 的定义：

```
static pte_t* walkpgdir(pde_t *pgdir, const void *va, int alloc)
```

首先获取页虚拟地址 va 所在页表在页表目录中的位置 PDX（va），然后在页表目录中获取该页表的入口信息(pde：页表入口信息=页表的地址+页表的状态)，&pgdir[PDX(va)]，然后，查看该页表是否存在（PTE_P），若存在，则从页表的入口信息中提取页表的地址（PTE_ADDR(*pde)），若不存在。则调用 kalloc 分配一页内存作为页表，进入 kalloc 的定义：

```
char* kalloc(void)
```

首先调用 acquire 获取 kmem 锁，然后将空闲页链表的头元素取出，最后调用 release 释放 kmem 锁，返回上述空闲页，kalloc 定义结束。

回到 walkpgdir 的定义，然后调用 memset 将新获得的空闲页内存清零，进入 memset 的定义：

```
void* memset(void *dst, int c, uint n)
```

调用 stosb，进入 stosb 的定义：

```
static inline void stosb(void *addr, int data, int cnt)
```

调用 x86 指令（cld；rep stosb）将 addr 地址及之后的 cnt 字节置为 data，stosb 定义结束。关于此定义中 gcc 内联汇编的语法，可自行 google（推荐 Brennan’s Guide to Inline Assembly）。

返回 memset 定义，返回 dst，memset 定义结束。

返回 walkpgdir 的定义，然后将 pgtab 的物理地址和一些标志位存到页表入口信息（pde），最后，获得虚拟地址 va 在 pgtab 中的位置，并将位置对应在 pgtab 中的的页入

口信息的地址返回，walkpgdir 定义结束。

返回 mappages 定义，检测获得的页入口信息的合法性，不合法就 panic，然后，设置每一页的页入口信息（包括物理页地址和标志位，权限等），mappages 定义结束。

返回 setupkvm 定义，最后，返回生成的页表目录地址，setupkvm 定义结束。

返回 kvmalloc 定义，接下来，调用 switchkvm 将内核页表目录地址载入页表目录基址寄存器，进入 switchkvm 的定义：

```
void switchkvm(void)
```

调用 lcr3 载入页表目录物理地址，进入 lcr3 定义：

```
static inline void lcr3(uint val)
```

调用 x86 指令将物理地址 val 载入页表目录基址寄存器，lcr3 定义结束。

回到 switchkvm 定义，switchkvm 定义结束。

返回 kvmalloc 定义，kvmalloc 定义结束。

mpinit()

此模块大量依赖 Intel Multi Processor Specification（xv6 用的是 1.4 版本，以下简称 Spec）中的内容，建议继续之前，大致浏览一下 Spec 的内容。

接下来，内核调用 mpinit 获取所有 cpu 信息，进入 mpinit 定义：

```
void mpinit(void)
```

首先，调用 mpconfig 获取 MP 配置表（MP Configuration Table），进入 mpconfig

的定义：

```
static struct mpconf* mpconfig(struct mp **pmp)
```

首先，调用 mpsearch 搜索 MP 浮点指针（MP Floating Pointer），进入 mpsearch 的定义：

```
static struct mp* mpsearch(void)
```

参考 Spec 38 页，了解 MP 浮点指针可能存在的 3 个位置，首先，到 BIOS 的 BDA 区域（位于物理地址 0x400 至 0x4ff）获取 EBDA 的基址（可 google “bios bda”获取详细的信息），EBDA 基址位于物理地址 0x40E，0x40F 两个字节中，将这两个字节左移 4 位得到真正的 EBDA 基址，若算出的 EBDA 基址存在(不等于 0)，则调用 mpsearch1 到 EBDA 基址开始的 1KB 内存中搜索 MP 浮点指针，进入 mpsearch1 的定义：

```
static struct mp* mpsearch1(uint a, int len)
```

在以 a 为起始物理地址，大小为 len 的内存区域中，搜索 MP 浮点指针数据结构（MP 浮点指针数据结构的判断依据：1.签名（头 4 个字节）为_MP_；2.所有字节之和为 0;），返回搜索到的 MP 浮点指针或返回 0，mpsearch1 定义结束。

回到 mpsearch 定义，若返回的 MP 浮点指针不为 0，则返回 MP 浮点指针，为 0，则继续到下一个位置搜索，接下来，若算出的 EBDA 基址不存在（等于 0），则调用 mpsearch1 到 system base memory(所谓 system base memory 是指内存中在 BIOS EBDA 前的 low memory 部分，其起始地址为 0x0）的最后 1KB（system base memory 的大小位于物理地址 0x413，0x414 两个字节中，将其乘以 1024 得到以字节为单位的 system base memory 的大小，由于 system base memory 的起始地址为 0x0，其末尾地址就是其大小）中搜索 MP 浮点指针，若返回的 MP 浮点指针不为 0，则返回 MP 浮点指针，为 0，则继续到下一个位置搜索，接下来来到最后一个位置，调用 mpsearch1 到 BIOS ROM 的 0xF0000（源码注释中有误，写的是 0xE0000）到 0xFFFFF 中搜索 MP 浮点指针，并将搜索结果返回，mpsearch 定义结束。

回到 mpconfig 定义，检测返回的 MP 浮点指针，若指针为 0 或 MP 浮点指针数据结构中的 MP 配置表地址为 0，则返回 0，接下来，从指针中获取 MP 配置表地址（conf），检测其合法性（配置表的签名（表的头 4 个字节）应为“PCMP”，表的版本为 1.4 或 1.1（version 成员为 1 或 4），表的所有字节的和应为 0），不合法就返回 0，最后，将 MP 浮点指针存到输入参数（pmp）中，返回 MP 配置表地址，mpconfig 定义结束。

回到 mpinit 定义，检测获取的 MP 配置表地址合法性，不合法就返回，接下来，从 MP 配置表中获取本地 APIC 基址，然后，进入各个表入口（Table Entry）获取相关信息（通过检测每个入口的第一个字节（Entry Type Code）来判断入口类型）：

- 进入处理器信息入口（MPPROC，每一个处理器都有一个入口，故多处理器设备上此 case 会进入多次），检测 APIC ID（apicid）是否从 0 开始连续递增（0，1，2，3，4.....，xv6 对于 APIC ID 不从 0 递增的设备，将其视为单 CPU 设备，但是，Spec 并没有规定 APIC ID 必须从 0 递增，只是规定其必须唯一，这或许是 xv6 为了使相关代码更简洁所做出的折中的决定），若否，将设备视为单 CPU 设备（ismp = 0），接下来，检测 cpu 是否是 bsp（bootstrap processor，启动 cpu），若是则将 bcpu 指向该 cpu，然后，设置各 cpu 的 id（设为 ncpu），最后，进入下一个入口；
- 进入 IO APIC 信息入口（MPIOAPIC），获取 IO APIC 的 ID（apicno 成员），进入下一个入口；
- 其余入口（MPBUS，MPIOINTR，MPLINTR：总线信息，IO Interrupt Assignment, Local Interrupt Assignment），不做处理，进入下一个入口；
- 若检测到的 Entry Type Code 不属于任何以上入口，则将设备视为单 CPU 设备；

接下来，设置单 CPU 设备的相关信息（ncpu，lapic，ioapicid），最后，检查 imcrp 位是

否被设置为 1，若是，则设置 0x22 和 0x23 位将模式从 PIC 切换到 APIC（只需将所有外部中断屏蔽即可，具体可参考“IA32 中断与异常研究”与 Spec 中相关内容），mpinit 定义结束。

lapicinit()

在继续阅读之前，建议先浏览 Manual 的第 10.1 节及之后相关章节和 APIC-OSDev Wiki 的相关内容。

接下来，内核调用 lapicinit 初始化本地 APIC，进入 lapicinit 的定义：

```
void lapicinit(void)
```

首先，检测本地 apic 基址是否存在，不存在就返回， 然后，设置伪中断向量寄存器 spurious interrupt vector register（SVR），使能 local APIC（SVR 的第 9 位--bit 8 置 1），并设置中断向量的数量（SVR 的低 8 位） （spurious-interrupt vector） （详细信息查看 Intel 64 and IA-32 Architectures Software Developer’ s Manual（以下简称 Manual）第 10.4.3 节和 APIC-OSDev Wiki 中的相关内容），接下来，设置计时器相关寄存器（详见 Manual 第 10.5.4 节内容）：

- 设置 Divide Configuration Register（TDCR），APIC 计时器的频率就是总线频率除以 TDCR 中的设置值（不是数值，有一个转换关系，详见 Manual 图 10-10），xv 设置的是 0xB（也就是 111，根据 Manual 中的转换关系转换后为 1，也就是 timer 的频率与总线频率相同）；

- 设置 LVT Timer Register（Timer），将计时器设置为周期性的（PERIODIC），这样一来，当 Current Timer Register（TCCR）递减到 0 时，会自动从 Initial Count Register（TICR）重新读取数值，将计时器的 Interrupt Vector 指向 IRQ_TIMER，这样一来，计时器计数到 0 时会产生一个指向 IRQ_TIMER 的中断；
- 设置 Initial Count Register（TICR）为 10000000，意为，以总线频率，起始值为 10000000 递减 TCCR，TCCR 到 0 时，产生一个指向 IRQ_TIMER 的中断，同时 TCCR 会自动从 Initial Count Register（TICR）重新读取数值（恢复为 10000000）；

接下来，关闭逻辑中断线，屏蔽 LINT0 和 LINT1，在多 CPU 设备中，LINT0 和 LINT1 不被用来将中断传给 CPU，而是将中断通过 IO APIC 传给 CPU，详见 Manual 第 8.7.13.4 节。然后，检测当前设备是否提供 Performance Counter Overflow Interrupts 入口，若提供，则屏蔽该 APIC 中断，详情可自行 google。然后，设置 IRQ_ERROR 中断向量。然后重置错误状态寄存器（Error Status Register，ESR），详见 Manual 第 10.5.3 节。接下来，重置中断结束寄存器（End Of Interrupt，EOI），中断程序在返回前必须设置 EOI，以便告知本地 APIC 中断处理已结束，便于其处理下一个中断，详见 Manual 第 10.8.5 节。然后，广播 Init Level De-assert 到所有本地 APICs 同步它们的 APIC ID 为仲裁 ID（Arbitration ID），详见 Manual 第 10.6.1 节和 10.7 节。最后，将任务优先寄存器值设置为最低（0），这样所有的中断都可以被 APIC 获取，详见 Manual 第 10.8.3.1 节。lapicinit 定义结束。

seginit()

在继续阅读以下内容之前，建议先浏览 Manual 的第 3.1 节及之后相关章节和“xv6-book”的 Appendix B，了解一下 x86 分段机制和 xv6 中对分段的处理。

接下来，内核调用 seginit 初始化段描述符，进入 seginit 的定义：

```
void seginit(void)
```

首先，调用 cpunum 获取当前 cpu 的 index，进入 cpunum 的定义：

```
int cpunum(void)
```

首先，调用 readeflags 获取当前 EFLAGS 寄存器的值，并查看其中的 IF（开中断位）是否被设置，若被设置且 cpunum 是第一次被调用，则打印相关信息，然后，检测 lapic 的存在性，不存在返回 0，存在就返回 APIC 的 ID（详见 Manual 的图 10-6 中的 APIC register，前面的章节提到过，APIC 的 ID 必须有一定规律，其可作为 cpus 的 index），cpunum 定义结束。

回到 seginit 定义，接下来，设置当前 cpu 的 gdt 各个段的描述信息（Segment Descriptor，xv6 对段的设置模式是 Basic Flat Model，也就是基本没有使用段的功能，详见 Manual 的 3.2.1 节，除了设置读，写，执行以及 DPL，其余都没有设置，详见 Maunal 图 3-8），内核和用户的代码段都是可执行和可读的，内核和用户的数据段都是可读可写（STA_W 的数值是 2 代表可读可写，详见 Manual 表 3-1）的，但内核段的 DPL（Descriptor Privilege Level）是最高的（0），而用户段的 DPL 是最低的（3），接下来，设置 CPU 段（SEG_KCPU）的描述信息，此段的 Base Address 指向当前 CPU 自身信息的数据结构（struct cpu，值得注意的是，此处设置的段长度为 8，因为这个段中不仅包含 cpu 的指针，还包含 proc 的指针，详见 struct cpu 的成员定义），然后，载入设置好的 gdt（lgdt），进

入 lgdt 的定义：

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

将 gdt 的大小减一存入 pd 第一个 16 位，接下来将 gdt 的地址分成两段存入 pd 的第 2 个和第 3 个 16 位，调用 x86 指令 lgdt 载入 pd 到 GDTR 寄存器（详见 Manual 的 2.4.1 节），lgdt 定义结束。

回到 seginit 定义，接下来，将当前 CPU 的段描述在 GDT 中的偏移载入到 gs 寄存器（loadgs），然后，给当前 cpu 数据结构指针（被规定是 gs 寄存器所指的段加上 0 偏移）赋值（就是上面初始化过的 c），最后，给当前 cpu 上运行的当前 proc（被规定是 gs 寄存器所指的段加上 4 偏移，因为 proc 指针在 cpu 结构体中紧随 cpu 指针之后，详见 struct cpu 的成员定义）赋值为 0（意为没有进程运行），这两行代码让每个 cpu 可以通过相同的代码获取自身信息（local cpu info）和运行于各个 cpu 上的进程信息（因为 seginit 会被每个 cpu 运行一次，每个 cpu 都将各自的 cpu 信息指针信息保存到各自的 gs 寄存器中，一定要注意，proc.h 中定义的 cpu 指针和 proc 指针在每个 cpu 上是不同的，对此 proc.h 明确注释道：各个 cpu 的%gs:0 就是&cpus[cpunum()]，%gs:4 就是 cpus[cpunum()].proc，这样一来每个 cpu 只需通过 cpu 指针和 proc 指针就能获取各自的 cpu 信息和 proc 信息，而不需每次都调用 cpunum()来获取 cpu index 再获取 cpu 和 proc 信息，同一套代码在不同的 cpu 上产生不同的效果，减少了代码的冗余）。seginit 定义结束。

cprintf()

接下来，内核调用 cprint 打印 bsp 的信息（cpu->id 在 mpinit 中定义）。

picinit()

接下来，内核调用 picinit 初始化中断控制器（pic，需要注意的是，pic 只在单 cpu 设备上才有用，多 cpu 设备上会直接将其屏蔽，使用 apic 来处理中断），进入 picinit 的定义：

```
void picinit(void)
```

此函数通过设置 io 端口来配置 pic，是 8259A 初始化的的标准流程，与 xv6 内核关系不大，故不深入讲解，有兴趣可以自行 google 其细节，picinit 定义结束。

ioapicinit()

在继续阅读之前，建议先浏览 Manual 的第 10.1 节及之后相关章节和 APIC-OSDev Wiki 的相关内容。

接下来，内核调用 ioapicinit 初始化 io apic，进入 ioapicinit 的定义：

```
void ioapicinit(void)
```

首先，检测当前设备是否为多 cpu 设备，若否则退出，接下来获取 ioapic 的地址(IOAPIC，ioapic 被 map 到了内存的 0xfec00000)，然后，获取重定向入口的最大数量 maxintr（maximum amount of redirection entries，其位于 0x01 寄存器的 bits 16-23)，然后，获取 IO APIC 的 ID（其位于 0x0 寄存器的 bits 24-27)，然后检测获取的 IO APIC 的 ID 是否与之前 MP 配置表中获取的 ID 相同，接下来，设置每一个中断（设置中断向量--入口的低 8 位，关中断--入口的 bit16（之后会打开)，将入口的高 32 位全部清 0，意为将中断发给 APIC ID 为 0 的 cpu，就是自己--bsp，详见上述 Wiki 中的 IO APIC Registers 部分)。ioapicinit 定义结束。

consoleinit()

在继续阅读之前，建议先浏览 Serial Uart--an in depth tutorial 和 Display Hardware，了解串口和显示相关信息。

接下来，内核调用 consoleinit 初始化终端设备，进入 consoleinit 的定义：

```
void consoleinit(void)
```

首先，初始化终端锁和输入锁，设置终端设备的读写函数，进入 consolewrtie 的定义：

```
int consolewrite(struct inode *ip, char *buf, int n)
```

首先解锁终端设备文件对应的 inode（iunlock 和 ilock 在以后的文件系统详解中一并

解释)，获取终端锁，将 buf 中的内容一个字节一个字节的通过 consputc 输出到终端，进入 consputs 的定义：

```
void consputc(int c)
```

首先，检测系统是否已崩溃（panicked），若是则关中断进入死循环，接下来，检测待输出的字符是否为退格符（效果是向左删除一个字符，xv6 使用 ctrl+H 表示或者 delete 键表示删除一个字符，后面的章节会详细说明），若是则调用 uartputc 向串口输出'\b',' ','\b' 三个字符产生左删的效果（为什么这样做，我没深入研究），若否则将向串口输出这个字符，进入 uartputc 的定义：

```
void uartputc(int c)
```

首先，检测串口是否存在（uart 的值），若否则返回，接下来，检测串口是否正在输出数据（Line Status Register 的 bit5 为无数据发送状态位，位于 COM+5），若是则循环检测最多 128 次，若否则输出字符到 THR（Transmitter Holding Register，位于 COM + 0），uartputc 定义结束。

回到 consputs 定义，接下来，调用 cgaputc 将字符在屏幕上打印出来，进入 cgaputc 的定义：

```
static void cgaputc(int c)
```

首先，获取当前光标的位置，先获得光标位置的高 8 位信息（光标位置高 8 位寄存器的 index offset--0xE（十进制是 14）写入 CRT 微控制器的 Data Register--port 0x3D4，然后从 CRT 微控制器的 Index Register--port 0x3D5 获取光标位置高 8 位寄存器的数值，详见 Display Hardware），然后，再获取光标位置的低 8 位信息（光标位置低 8 位寄存器的 index offset--0xF（十进制是 15）），接下来，若待打印的字符是换行符（'\n'），则将光标移到下一行的开头（设置的显示模式应该是 80x50），若待打印的字符是退格符（BACKSPACE），则将光标左移一个字符，其余情况，就将字符加上属性打印出来（xv6 使用的 CGA 显示，其映射到内存的地址是 0xB8000-0xBFFFF，其使用 2 个 byte 代表一个

字符，低 8 位为字符的 ascii 码，高 8 位为显示属性，在此，高 8 位被设置为 0x07，意为前景是黑色，背景是白色，详见 Display Hardware)，接下来，检测打印完当前字符后，是否需要 scroll（屏幕满了，必须要将所有显示内容整体上移一行)，若是则将显存的第一行（头 80 个字符）删除，并将其余行前移一行，最后，刷新光标的位置并将光标当前位置显示为一个黑色的空格。cgaputc 定义结束。

返回 consputs 的定义，consputs 定义结束。

返回 consolewrtie 的定义，接下来，释放终端锁，给终端设备文件 inode 加锁，返回输出的字符数。consolewrtie 定义结束。

回到 consolecinit 定义，进入 consoleread 的定义：

```
int consoleread(struct inode *ip, char *dst, int n)
```

首先，查看是否还有未读的 buffer（input.w >input.r）,若否则循环等待（等待过程中进程被杀则返回)，若是，则将 buffer 读出，若读出的字符为 EOF 或换行符（'\n')，则完成读取，返回读取的字符数，consoleread 定义结束。

回到 consolecinit 定义，然后，使能终端锁 (locking=1 为使能，并不是上锁)，最后，打开键盘中断（单 cpu 设备调用 picenable，多 cpu 设备调用 ioapicenable，并将中断绑定到 bsp--APIC ID 为 0)。consoleinit 定义结束。

uartinit()

在继续阅读之前，建议先浏览 Serial Uart--an in depth tutorial (以下简称 tutorial)，了解串口的相关信息。

接下来，内核调用 uartinit 初始化串口，进入 uartinit 的定义：

```
void uartinit(void)
```

首先，关闭 FIFO（COM1 的端口地址是固定的 0x3F8，com1+2 是 Interrput Identification Register，其高 2 位置 0 表示不使用 FIFO，由于不同型号的串口的 FIFO 设置可能不同，会影响读写串口的机制，进而增加内核代码的复杂度，关闭 FIFO，就可以用同一套代码处理不同型号的串口，详见 tutorial)，接下来，置 DLAB 为 1（com1+3 是 Line Control Register，其 bit7 为 DLAB 开关，将其置为 1，是为了接下来写 DLL 和 DLM 来设置波特率，详见 tutorial)，然后，设置波特率为 9600（在 DLAB 置 1 时，com1+0 和 com1+1 分别是 DLL 和 DLM 寄存器，同时写这两个寄存器来设置串口的波特率，详见 tutorial)，接下来，置 DLAB 为 0，并设置字宽为 8 位（DLAB 置 0 是因为完成了波特率设置，接下来要使用 RBR 和 THR 等寄存器来收发数据，字宽为 8 位，意为每发送 8 个位就发间隔位，详见 tutorial)，然后，将 MCR 清零（Modem Control Register，用来与串口连接设备握手，详见 tutorial)，接下来，使能中断（com1+1 的 Interrupt Enable Register 置 1，意为 UART 可以主动发送中断给 cpu，详见 tutorial)，然后，检测串口是否存在（com1+5 的 Line Status Register 为 0xFF，意为串口不存在？)，不存在就返回，然后，清空 IIR 和 RBR，使能串口中断，最后打印一些信息到串口，uartinit 定义结束。

pinit()

接下来，内核调用 pinit 初始化进程表，进入 pinit 的定义：

```
void pinit(void)
```

只是初始化了进程表的锁，pinit 定义结束。

tvinit()

接下来，内核调用 tvinit 初始化中断描述表，进入 tvinit 的定义：

```
void tvinit(void)
```

首先，初始化中断描述表（中断向量的生成过程，后面的章节会讲到？），最后，初始化 ticks 锁（每一个 TIMER 中断，ticks 都会自增），tvinit 定义结束。

binit()

接下来，内核调用 binit 初始化磁盘缓存，进入 binit 的定义：

```
void binit(void)
```

首先，初始化磁盘缓存锁，然后，将 10 块磁盘缓存组成链表，binit 定义结束。

fileinit()

接下来，内核调用 fileinit 初始化文件表，进入 fileinit 的定义：

```
void fileinit(void)
```

只是初始化文件表的锁，fileinit 定义结束。

iinit()

接下来，内核调用 iinit 初始化 inode 缓存，进入 iinit 的定义：

```
void iinit(void)
```

只是初始化 inode 缓存的锁，iinit 定义结束。

ideinit()

接下来，内核调用 ideinit 初始化磁盘设备，进入 ideinit 的定义：

```
void ideinit(void)
```

首先，初始化磁盘锁，接下来，使能磁盘中断（在多 cpu 设备上，将磁盘中断绑定到最后一个 cpu 上--cpu id 最大的那个 cpu），然后，调用 idewait 等待磁盘启动完成，进入 idewait 的定义：

```
static int idewait(int checkerr)
```

继续阅读前，建议先浏览下 IO_devices 和 ATA PIO Mode 文档，了解下 ide。

首先，轮询磁盘状态寄存器，直到磁盘状态为 READY（磁盘状态寄存器的端口地址是 0x1F7，其中的 bit6 是 READY 位，bit7 是忙位，详见 IO_devices），若参数 checkerr 为 1，则查看磁盘状态是否有错误（磁盘状态寄存器的 bit0 和 bit5 是错误位），有错返回-1，其余返回 0。idewait 定义结束。

返回 ideinit 定义，接下来，检测设备是否存在第二块磁盘分区（disk0 是第一块，disk1 是第二块），将端口 0x1f6 置 0xf0（0xe0 | (1<<4)）来选择 disk1，等待 0x1f7 的值变化成非 0 值（非 0 表示 disk1 存在），若 0x1f7 依然是 0，则 disk1 不存在，最后恢复到 disk0 的选择（将端口 0x1f6 置 0xe0）。ideinit 定义结束。

timerinit()

接下来，内核调用 timerinit 初始化单一 cpu 设备的计时器，进入 timerinit 的定义：

```
void timerinit(void)
```

不想深入研究 intel 8253 微控制器的操作，只需知道该函数将设置 xv6 的计时器中断频率为 100 次/秒，整个 timer.c 文件可以作为一个黑盒模块看待，因其唯一的接口就是 timerinit，timerinit 定义结束。

startothers()

接下来，内核调用 startothers 初始化其余 CPU（若是多 cpu 设备），进入 startothers 的定义：

```
static void startothers(void)
```

首先，将 entryother.S 的二进制镜像载入到 0x7000 地址上，然后，依次初始化各个 CPU：

- bsp 不用再启动（c==cpus+cpunum()）；
- 给每个 CPU 分配栈空间，并将栈地址保存到 0x7000-32（code-4）；
- 每个 CPU 的 0x7000-64（code-8）都指向 mpenter（ap 启动后跳转到此处继续

执行)；

- 每个 CPU 都使用 entrypgdir 作为页目录；
- bsp 调用 lapicstartap 发送启动指令给各个 cpu；
- bsp 等待每个 cpu 启动完成，继续启动下一个 cpu；

进入 mpenter 的定义：

```
static void mpenter(void)
```

首先，调用 switchkvm 使用 bsp 已设置好的内核页表（所有 CPU 都共享这个页表），然后，每个 CPU 都初始化各自的段描述符 (除了 SEG_KCPU 部分不同，其余部分都相同)，接下来，每个 CPU 初始化 local APIC，最后，调用 mpmain 进入正常运行状态。mpenter 定义结束。

回到 startothers 定义，startothers 定义结束。

kinit2(P2V(4*1024*1024)，P2V(PHYSTOP))

接下来，内核调用 kinit2 初始化剩余的内存，进入 kinit2 的定义：

```
void kinit2(void *vstart, void *vend)
```

首先，将范围内的内存逐页初始化并添加到内核空闲内存链表，最后，启用内核内存锁（因为可能有多个 cpu 同时使用内核内存)。kinit2 定义结束。

userinit()

接下来，内核调用 userinit 创建第一个用户进程，进入 userinit 的定义：

```
void userinit(void)
```

首先，调用 allocproc 创建一个进程句柄，进入 allocproc 的定义：

```
static struct proc* allocproc(void)
```

首先，从进程表中找一个空闲的进程结构体（struct　proc），找不到就返回 0，找到了就初始化它：

- 将进程状态设置为孵化态（正在初始化）；
- 设置进程号（pid）；
- 给进程分配内核栈，分配失败就返回 0；
- 在内核栈中设置 trapframe；
- 在内核栈中设置 trapret；
- 在内核栈中设置寄存器；
- 设置程序计数寄存器（eip）指向 forkret；

这样设置后，未来该进程被进程调度器启动时，会依次执行 forkret（跳转到 trapret），trapret（从 trapframe 中恢复用户寄存器），最后回到用户空间。allocproc 定义结束。

回到 userinit 定义，接下来，给进程创建页目录（调用 setupkvm，创建并初始化其内核部分），然后，设置页表的用户部分（inituvm），进入 inituvm 的定义：

```
void inituvm(pde_t *pgdir, char *init, uint sz)
```

首先，分配一页内存用来装载 initcode（initcode 由 initcode.S 汇编而来）程序，接

下来，将这页内存映射到虚拟地址 0x0（一个进程的程序镜像开始于进程虚拟地址 0x0），并将这个映射保存到页表（mappages），最后将 initcode 程序装载到内存页。inituvm 定义结束。

回到 userinit 定义，接下来，设置进程占用的内存大小（sz），然后设置进程的 trapframe：

- 设置 tf 的 cs 的段类型为 SEG_UCODE；
- 设置 tf 的 ds，es，ss 的段类型为 SEG_UDATA；
- 打开 eflags 的中断使能（FL_IF）；
- 设置 esp（栈指针）指向程序镜像的末尾；
- 设置 eip（指令指针）指向程序镜像的开头；

接下来，设置进程名，进程的工作目录，最后，将进程的状态设置为 RUNNABLE（可被进程调度器调度执行），userinit 定义结束。

mpmain()

接下来，内核调用 mpmain 完成内核启动的最后一个步骤，进入 mpmain 的定义：

```
static void mpmain(void)
```

每一个 cpu 都会调用这个函数来完成最后的设置，首先，调用 idtinit 装载中断向量表，接下来，将 cpu 状态设置为已启动（started，bsp 在等待这个标志位），最后，启动进程调度器 scheduler，进入 scheduler 的定义：

```
void scheduler(void)
```

每一个 cpu 都运行有一个 scheduler，首先，允许中断，然后，从进程表中找到第一个状态为 RUNNABLE 的的进程，接下来，载入这个进程的虚拟内存（switchuvm），进入 switchuvm 的定义：

```
void switchuvm(struct proc *p)
```

首先，设置任务状态段（SEG_TSS）,当进程被中断时，会使用设置的栈（栈的段位于 SEG_KDATA，栈的指针位于 kstack），然后，将 SEG_TSS 载入任务寄存器（ltr），最后，载入进程的页目录。switchuvm 定义结束。

回到 scheduler 定义，接下来设置进程状态为 RUNNING，然后，保存当前各寄存器值，载入进程的寄存器值（swtch），至此，进程开始执行，到时间片用完时，其主动放弃 cpu 占用，cpu 重新回到 scheduler，继续执行 switchkvm，重新载入内核页表，进入下一个调度循环。scheduler 定义结束。

回到 mpmain 定义，mpmain 定义结束。

至此，内核的初始化程序结束，系统进入正常运行状态。