XV6 Boot e Mem Manage

Sistemas Operacionais
2017-1
Flavio Figueiredo (http://flaviovdf.github.io)

Boot

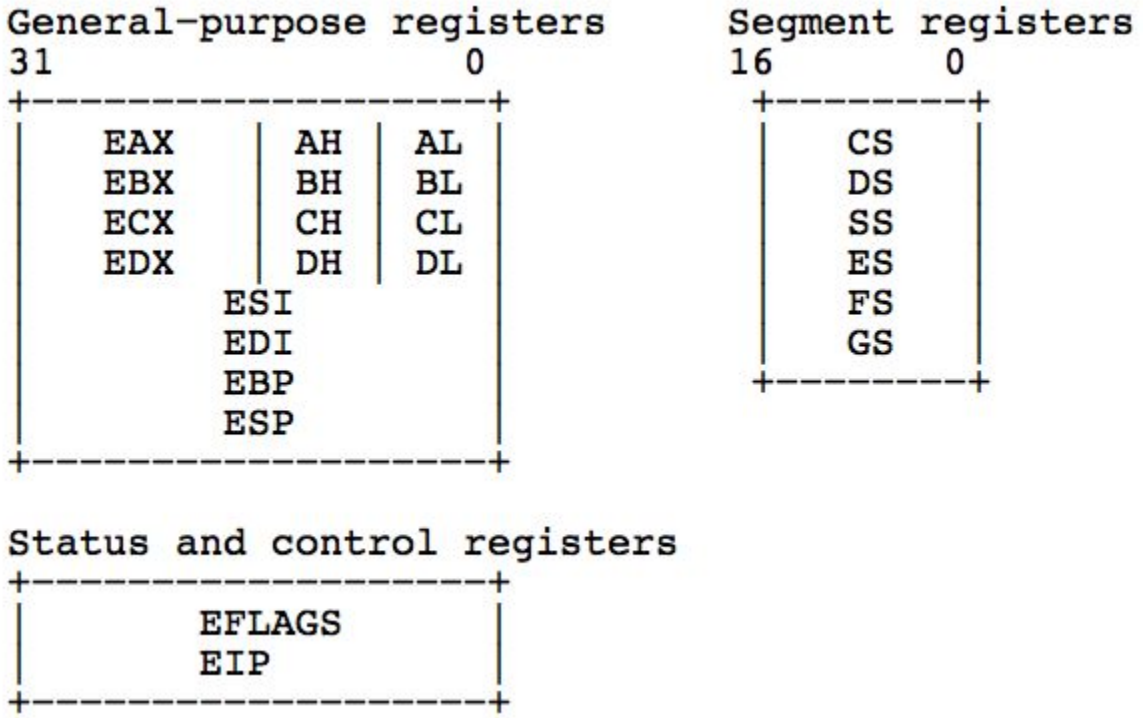
Até agora...

- Entendemos sobre processos
- Sabemos o básico de gerência de memória
- Vamos utilizar isto para mostrar o processo de Boot de XV6
- Vamos olhar nas rotinas de memória também

XV6

- O XV6 é um Unix simplificado
- Funciona apenas com 32 bits
  - Não vamos ver estruturas avançadas para gerenciar memória
- QEMU cuida de emular em uma máquina 64 bits
  - Aula de virtualização

Alguns Registradores do xv86



Boot

1. Precisamos ler um bootloader do *master boot record*
   a. Quando você instala o Windows/Linux/Mac o mesmo escreve um pequeno código em um local fixo do disco
   b. Exemplo com fdisk

```
Command (m for help): p
Disk /dev/sda: 465.8 GiB, 500107862016 bytes, 976773168 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 4096 bytes
I/O size (minimum/optimal): 4096 bytes / 4096 bytes
Disklabel type: dos
Disk identifier: 0x49c7f1a0

Device     Boot Start      End   Sectors   Size Id Type
/dev/sda1  *         2048 960194559 960192512 457.9G 83 Linux
/dev/sda2       960196606 976771071  16574466   7.9G  5 Extended
/dev/sda5       960196608 976771071  16574464   7.9G 82 Linux swap / Solaris
```

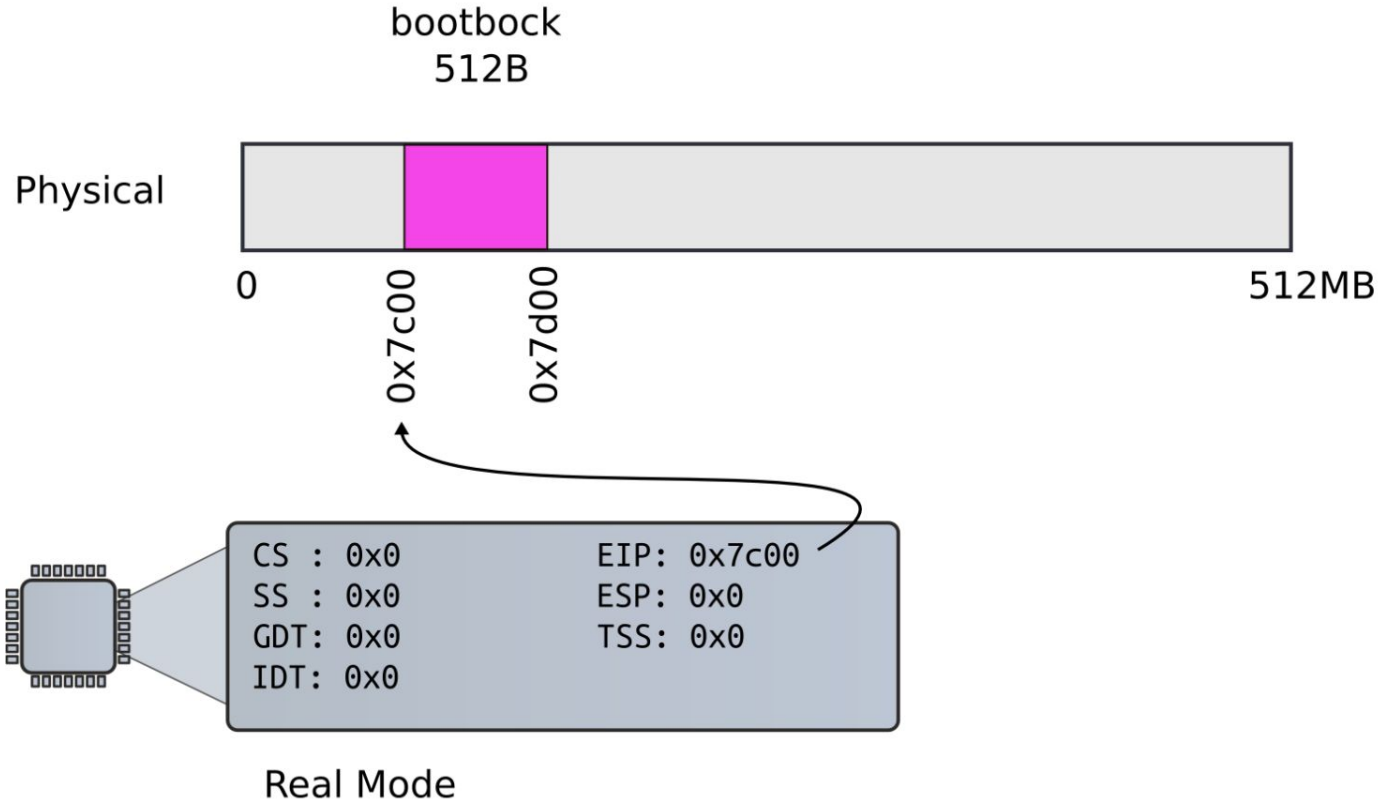
Passo 0

- Carregar código de boot na memória

EIP

- Extended Instruction Pointer

EIP


bootbock
512B
Physical
0
0x7c00
0x7d00
512MB
CS : 0x0
SS : 0x0
GDT: 0x0
IDT: 0x0
EIP: 0x7c00
ESP: 0x0
TSS: 0x0
Real Mode

Boot Passos Iniciais

- Desabilitar Interrupções
  - Não temos ninguém para tratar as mesmas
  - Não registramos endereços de tratamento
  - A bios pode tratar interrupts, não queremos isto

- Zerar segmentos

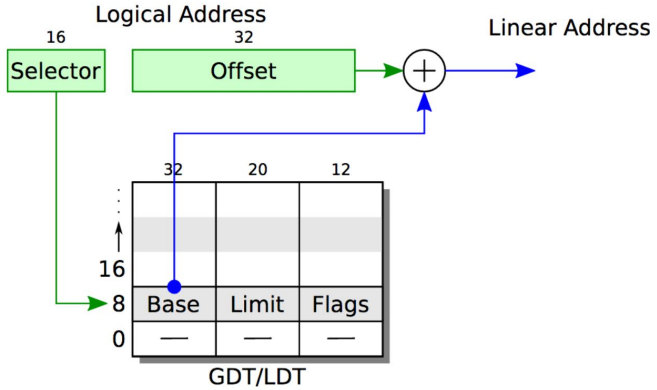
```
# Zero data segment registers DS, ES, and SS.
xorw    %ax,%ax             # Set %ax to zero
movw    %ax,%ds             # -> Data Segment
movw    %ax,%es             # -> Extra Segment
movw    %ax,%ss             # -> Stack Segment
```

GDT e Paginação

- Lembre-se que o hardware da suporte para paginação
- Não queremos fazer uso dela ainda no boot
- Desabilitamos

```
# Switch from real to protected mode.  Use a bootstrap GDT that makes
# virtual addresses map directly to physical addresses so that the
# effective memory map doesn't change during the transition.
lgdt    gdtdesc
```

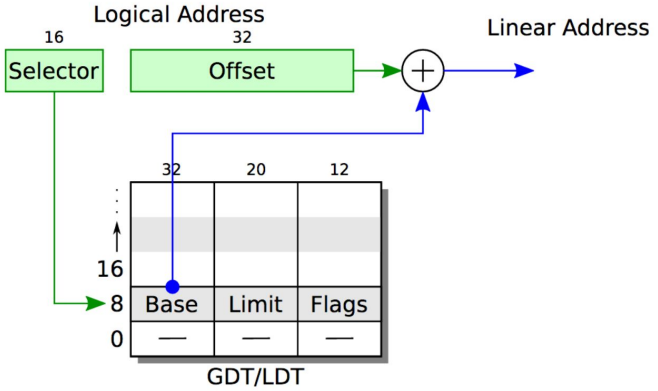
gdtdesc



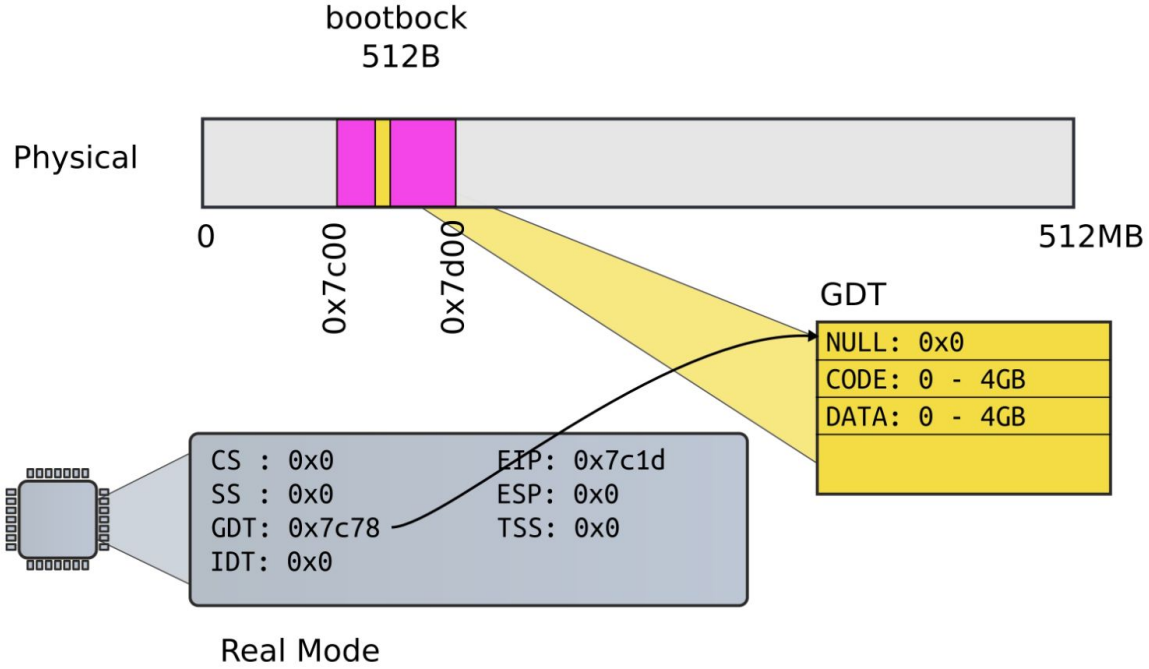

```
# Bootstrap GDT
.p2align 2
gdt:
  SEG_NULLASM                              # null seg
  SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)    # code seg
  SEG_ASM(STA_W, 0x0, 0xffffffff)          # data seg

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

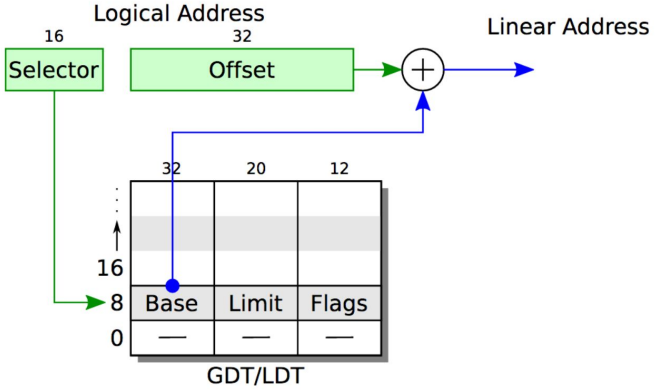
Basicamente

- Zeramos a base
- Limite máximo



Basicamente



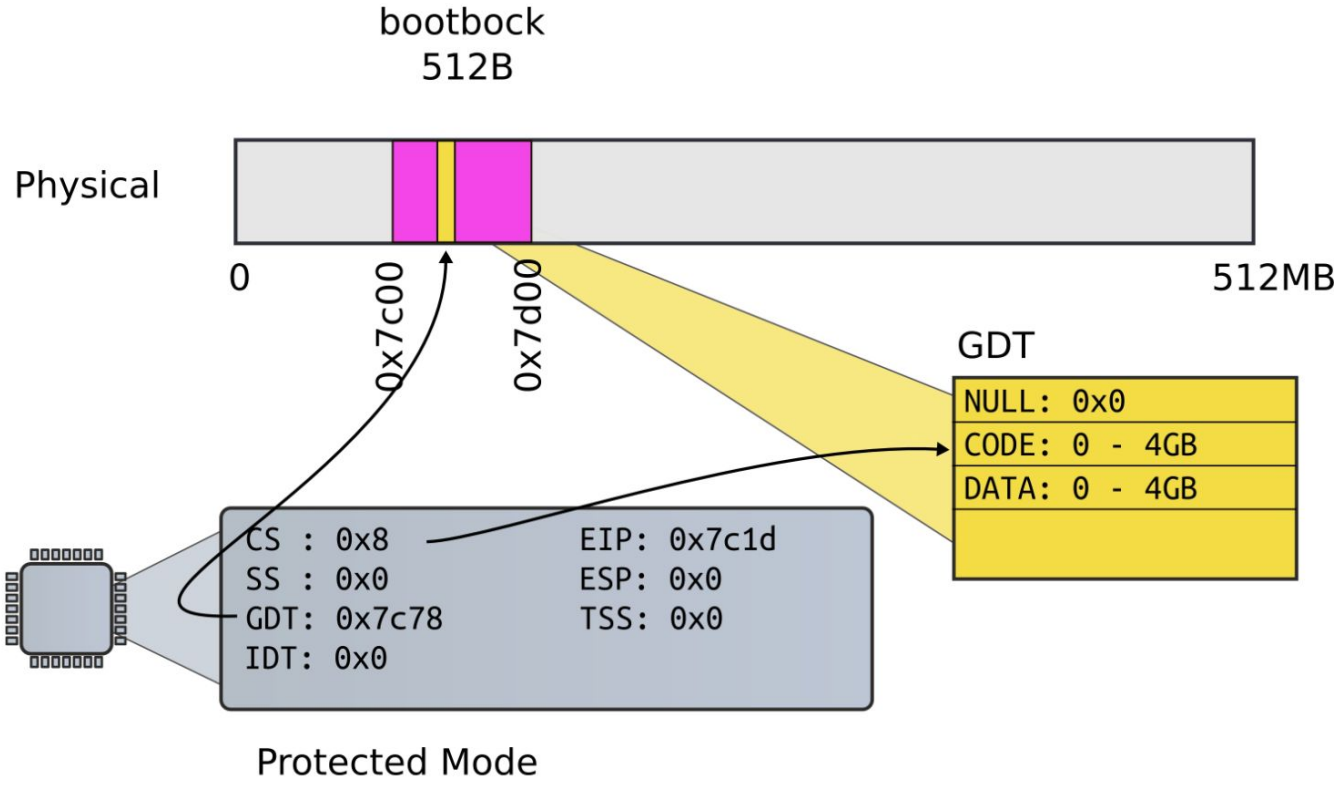


Jump para código de boot

- start32
  - Detalhes omitidos: iniciamos em um ambiente de 16 bits

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

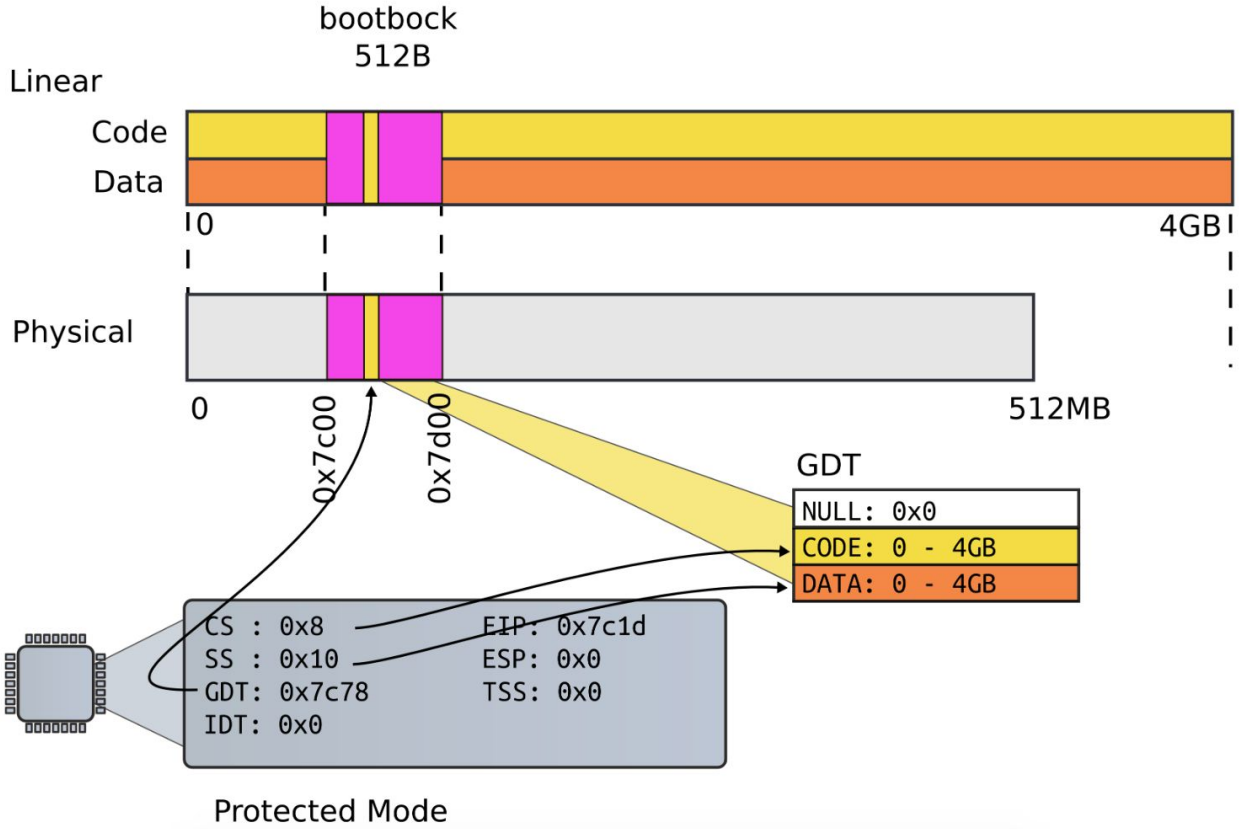
Jump para código de boot



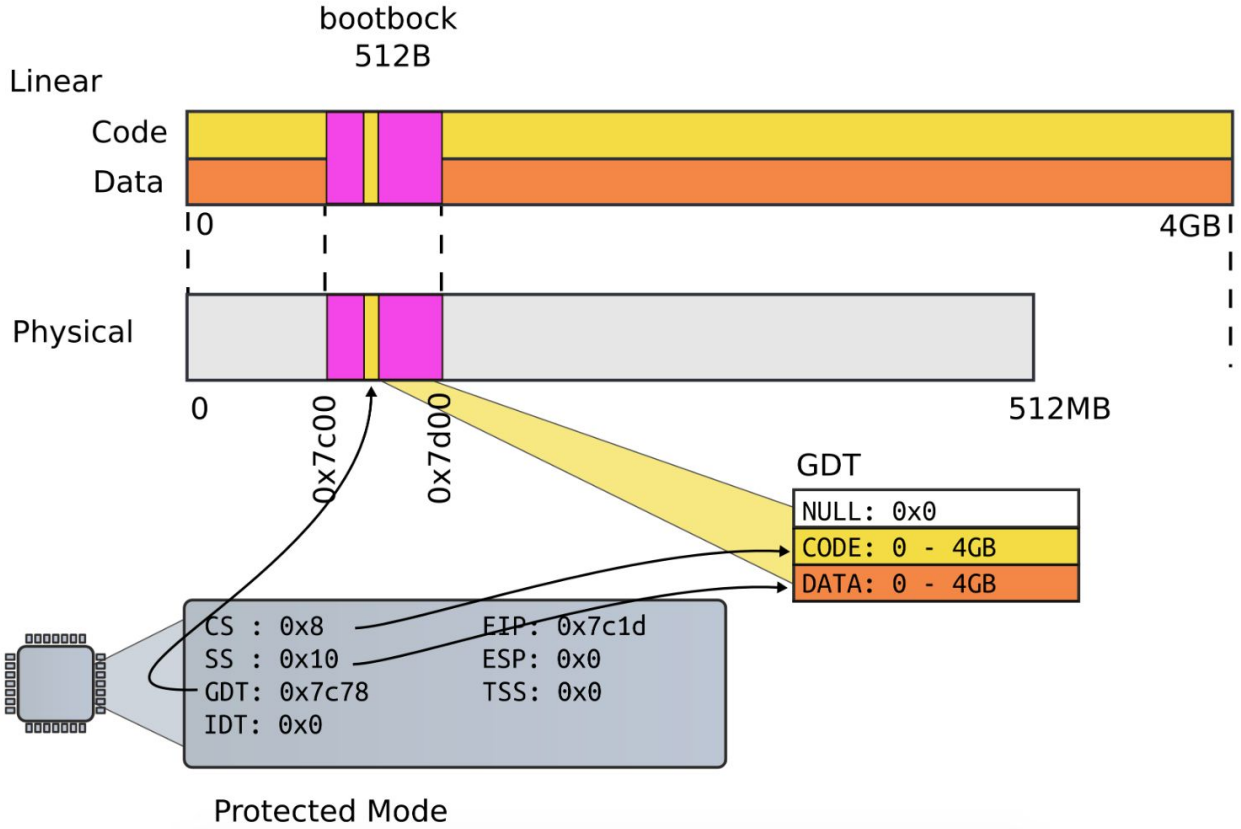
Atualizamos Segmentos

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

Atualizamos Segmentos

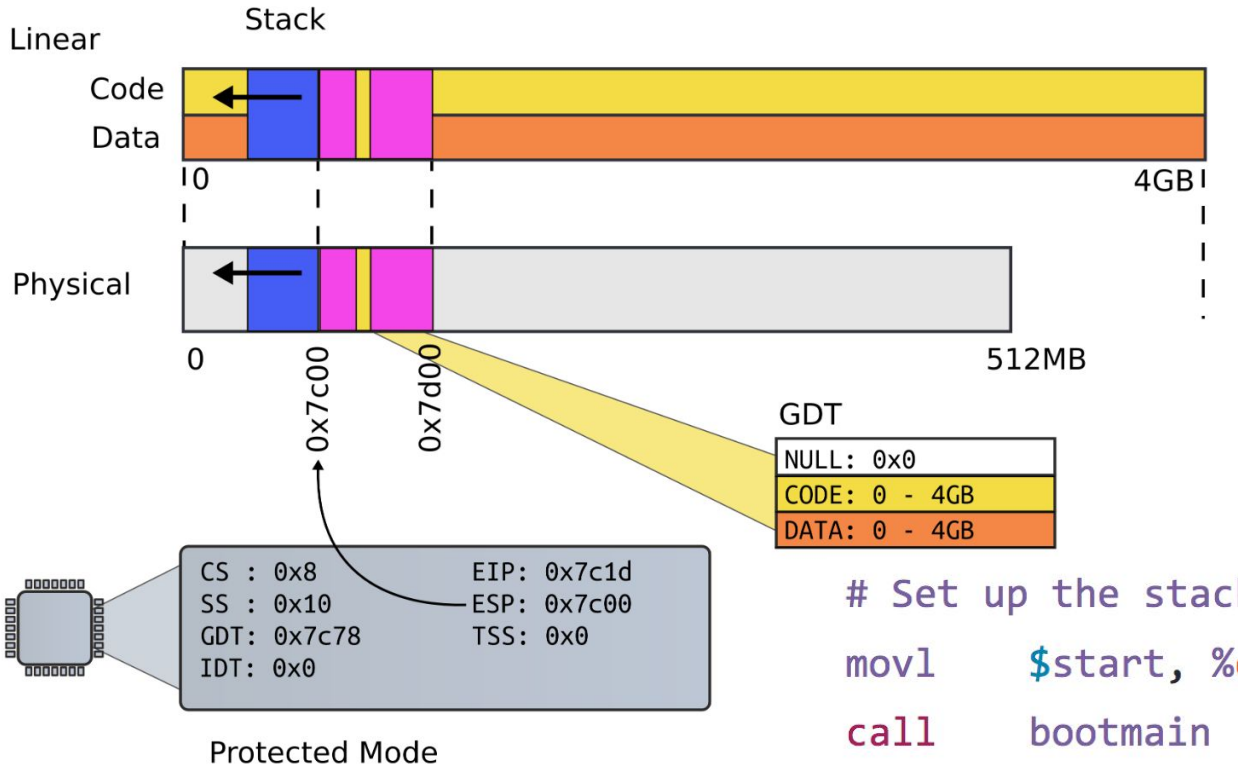


Atualizamos Segmentos



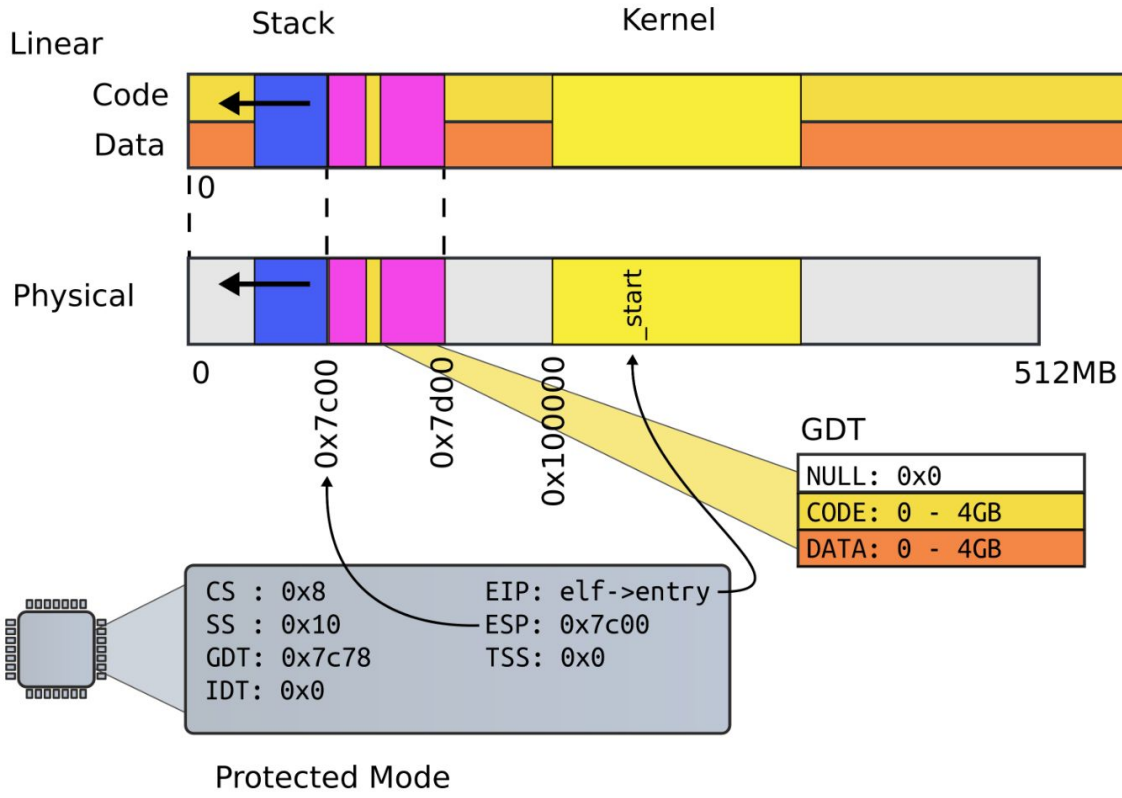
Basicamente temos uma bagunça onde os segmentos código e dados ocupam 4GB. Independente do tamanho da memória

Precisamos de uma pilha (entrar em código C)




```
# Set up the stack pointer and call into C.
movl    $start, %esp
call    bootmain
```

Carregamos o código do kernel

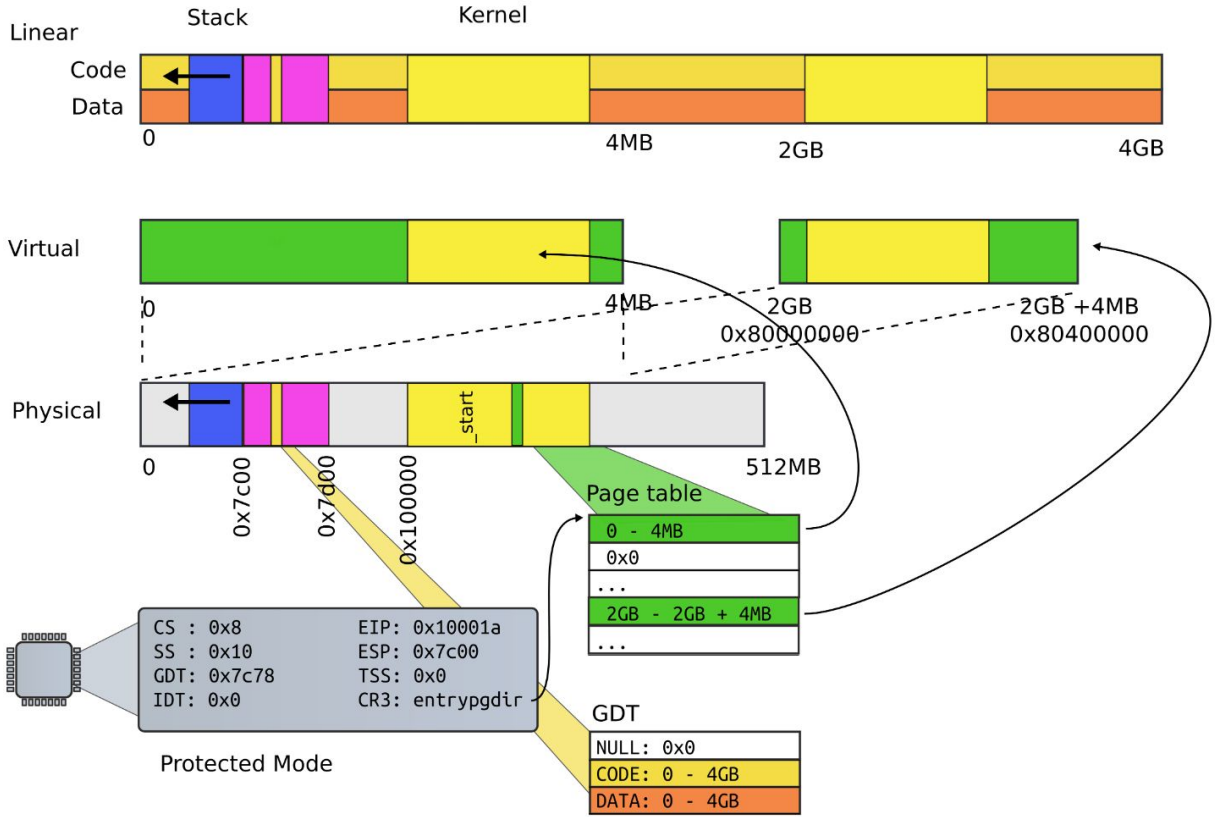



```
void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
  uchar* pa;

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

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

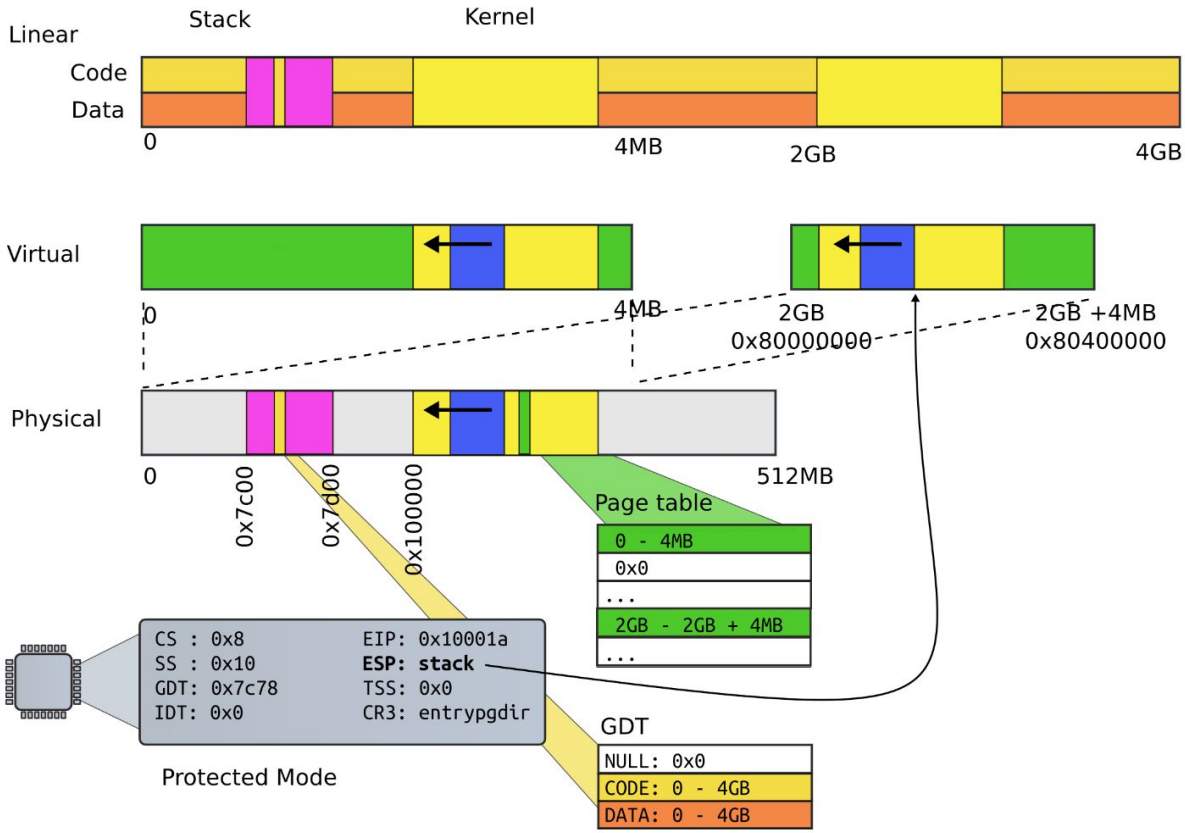
Detalhes Omitidos (vamos focar no essencial)

- **Alocação de algumas páginas iniciais para boot**
  - **Precisamos de alguma páginas para realmente ligar a MMU**
  - **Após isto, saímos do GDT sem tradução**
  - **2 páginas de 4MB para isto (código inicial do kernel)**
- + detalhes de como atualizar segmentos
- De qualquer forma, chegamos no kernel
- Podemos vamos main!

Após iniciar páginas iniciais



Com o stack chamamos main



Credits

https://www.ics.uci.edu/~aburtsev/

Anton Burtsev
University of California Irvine

Ufa!

- Agora o kernel pode tomar conta de tudo
- Iniciar gerência de memória
- Escrever quem cuida de traps e interrupts
  - Posições fixas da memória que o hardware também sabe

    https://github.com/mit-pdos/xv6-public/blob/master/traps.h

    https://github.com/mit-pdos/xv6-public/blob/master/trapasm.S
- Iniciar outros processos....
  - Precisamos de paginação para isto

main

- Primeiras 2 linhas memória
- Interrupções
  e
  Traps
- Arquivos e disco

...

```
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // detect other processors
  lapicinit();     // interrupt controller
  seginit();       // segment descriptors
  cprintf("\ncpu%d: starting xv6\n\n", cpunum());
  picinit();       // another interrupt controller
  ioapicinit();    // another interrupt controller
  consoleinit();   // console hardware
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  ideinit();       // disk
  if(!ismp)
    timerinit();   // uniprocessor timer
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  mpmain();        // finish this processor's setup
}
```

XV6 Memory

x86 Memory

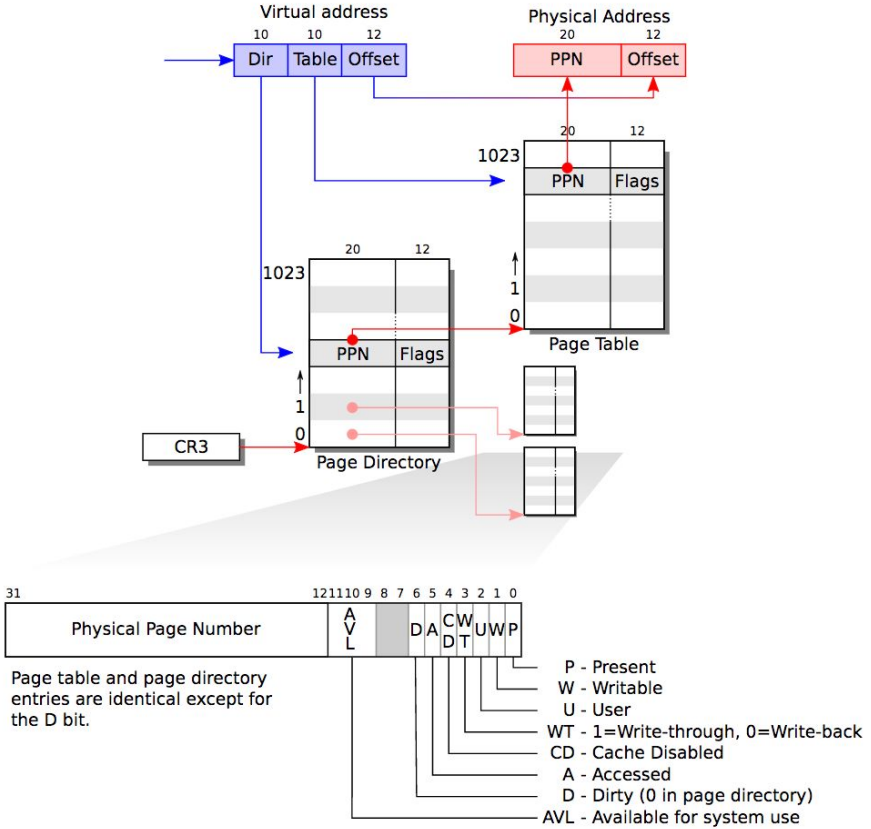
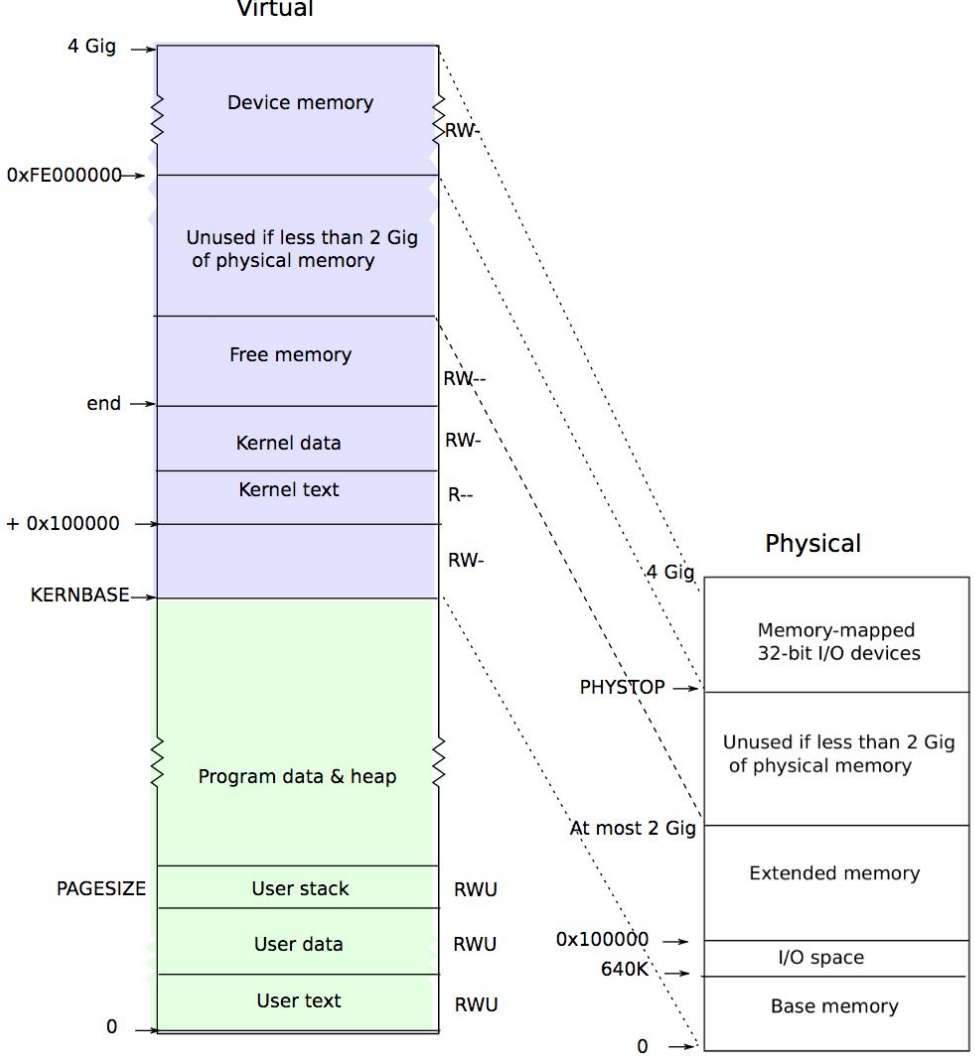


Figure 2-1. x86 page table hardware.

XV6 Layout


Virtual
4 Gig
Device memory
RW-
0xFE000000
Unused if less than 2 Gig of physical memory
Free memory
RW--
end
Kernel data
RW-
Kernel text
R--
+ 0x100000
RW-
KERNBASE
Program data & heap
PAGESIZE
User stack
RWU
User data
RWU
User text
RWU
0
Physical
4 Gig
Memory-mapped 32-bit I/O devices
PHYSTOP
Unused if less than 2 Gig of physical memory
At most 2 Gig
Extended memory
0x100000
I/O space
640K
Base memory
0

XV6 Layout

- https://github.com/mit-pdos/xv6-public/blob/master/memlayout.h
- https://github.com/mit-pdos/xv6-public/blob/master/mmu.h
- https://github.com/mit-pdos/xv6-public/blob/master/proc.h
- https://github.com/mit-pdos/xv6-public/blob/master/vm.c
- https://github.com/mit-pdos/xv6-public/blob/master/kalloc.c

XV6

- Tabelas de páginas livres com listas encadeadas
- 2 níveis
- Tradução em 2 passos por causa do x86
- https://pdos.csail.mit.edu/6.828/2016/xv6/book-rev9.pdf

setupkvm

- Inicia a tabela de páginas do SO
- Mapeia um espaço fixo da memória física
- Inicialmente apenas o espaço do kernel
  - Fixo no XV6

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

setupkvm

```
// Set up kernel part of a page table.
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (P2V(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0)
      return 0;
  return pgdir;
}
```

switchkvm

- Passa a utilizar a tabela de páginas na memória do kernel
- Basicamente atualiza o CR3

X86

- Registrador CR3 mantém onde inicia a tabela de páginas do processo
- Hardware de MMU pode atualizar bits de páginas
  - TLB

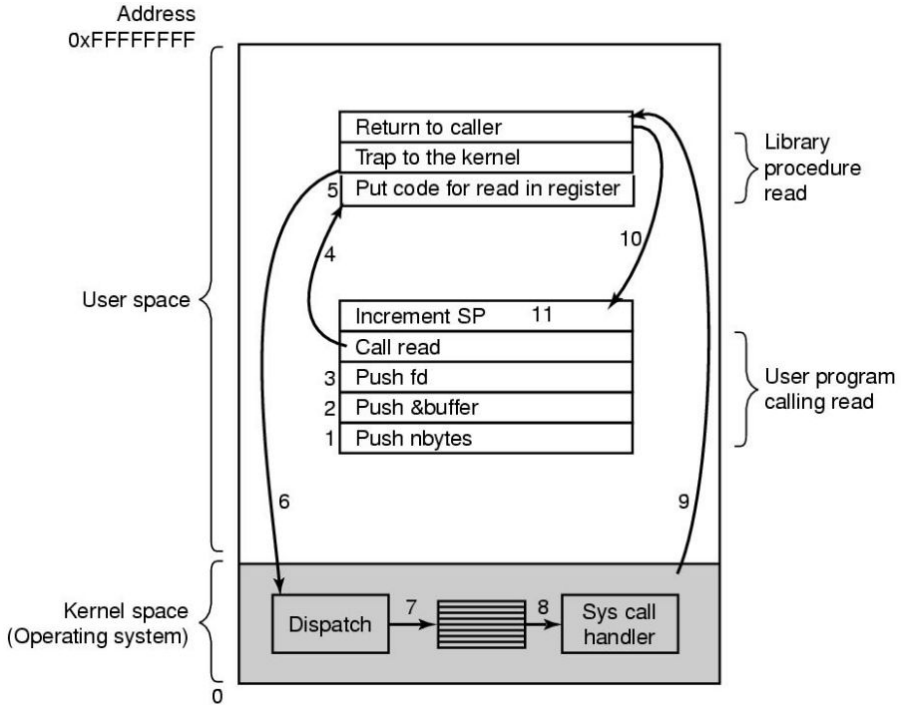
kinit2

- Mapeia o resto da memória na tabela de páginas

Precisamos de um init

- Até agora estamos em no nível privilegiado
- Criamos um processo init inicial
- Após isto
  - fork
  - exec
- userinit
  - 1 init por CPU no XV6
- A partir daqui podemos escalonar
- Podemos sair do modo privilegiado (setar 1 bit apenas)

Como voltar para o modo privilegiado?



Pequeno Desvio

- Saimos do percurso do livro para falar do Boot do XV6
- Dar uma ideia de como iniciamos a tabela de página