143A: Principles of Operating Systems

Lecture 10: Address spaces (Kernel memory allocator)

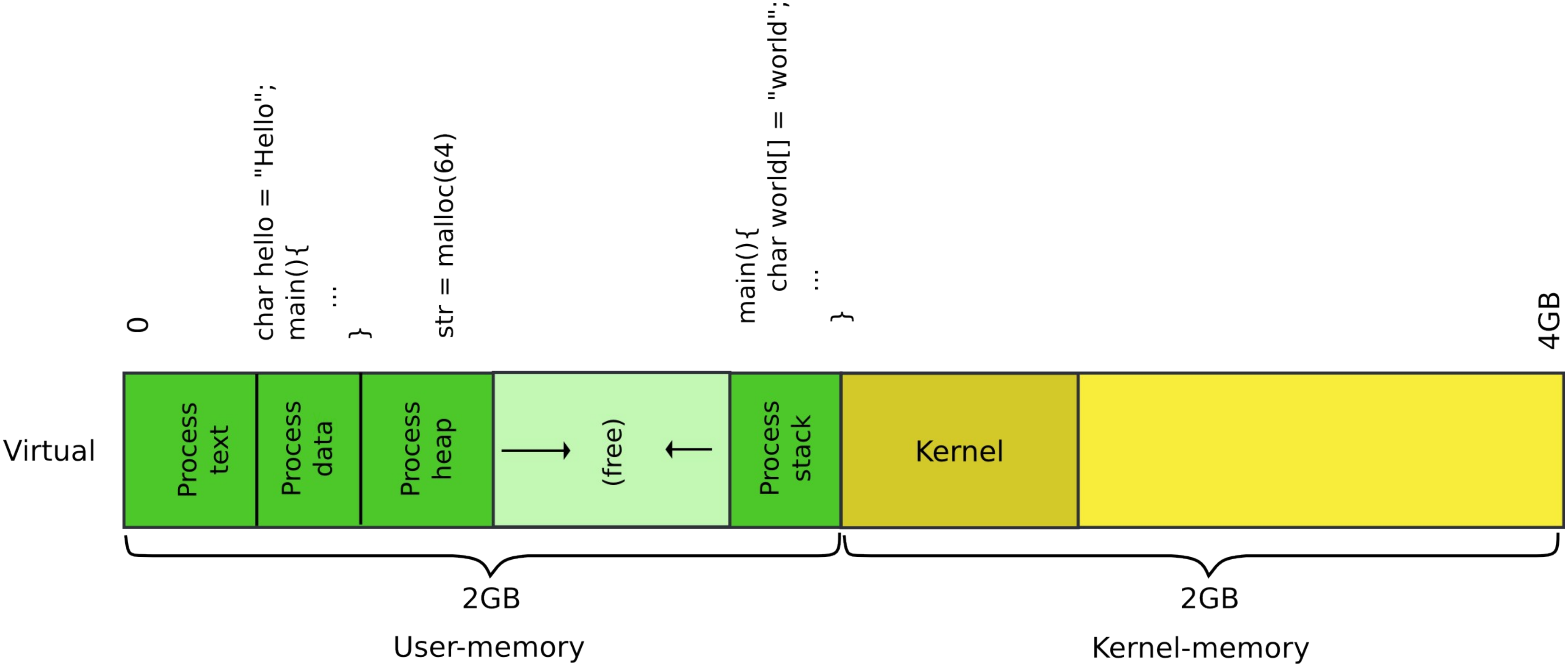
Anton Burtsev
October, 2017

Recap of the boot sequence

- Setup segments (data and code)
- Switched to protected mode
  - Loaded GDT (segmentation is on)
- Setup stack (to call C functions)
- Loaded kernel from disk
- Setup first page table
  - 2 entries [0 : 4MB] and [2GB : (2GB + 4MB)]
- Setup high-address stack
- Jumped to main()

What's next?

Our goal: 2GB/2GB address space



- Kernel needs normal 2 level page table
    - Right now we have only two entries
    - And current page table is only good for 4MB pages

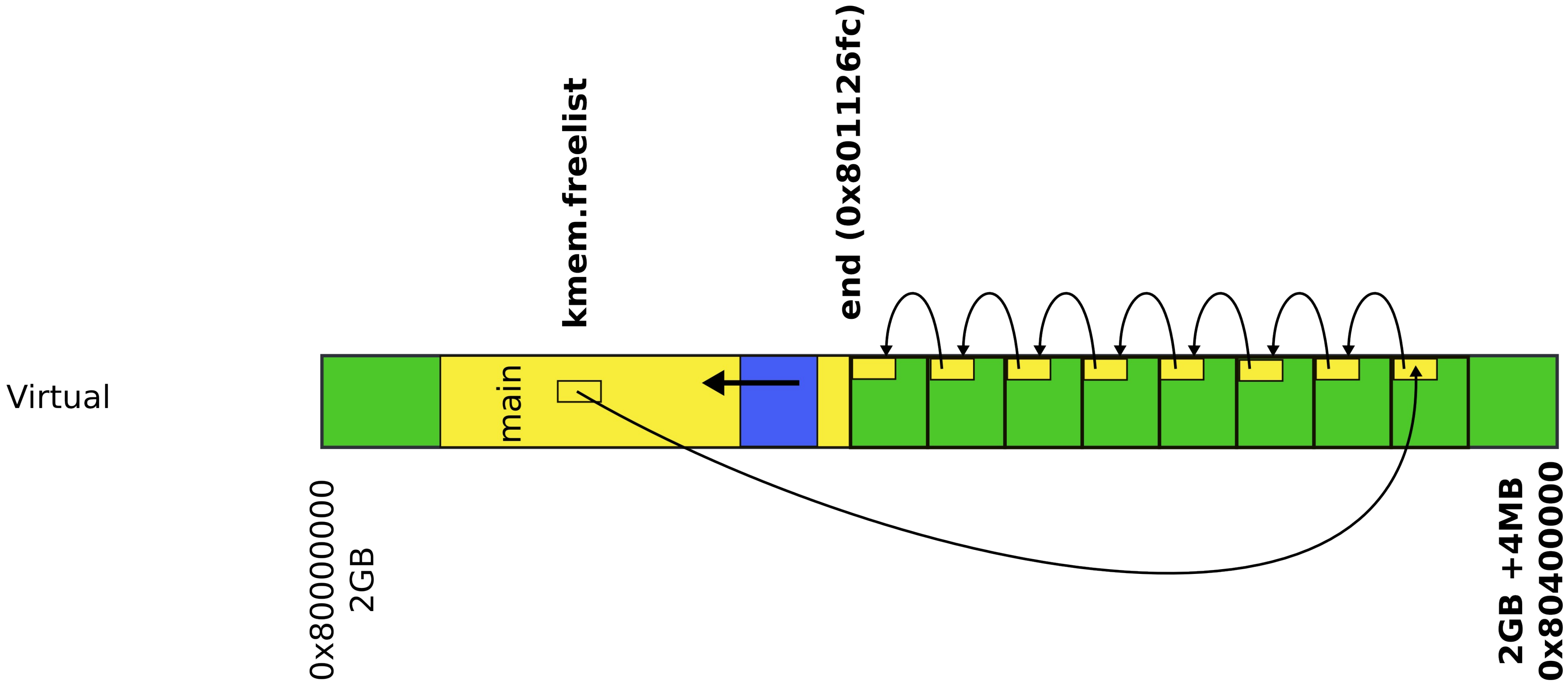
- But to create page tables we need memory
    - Where can it come from?

Physical page allocator

- Goal:
  - List of free physical pages
  - To allocate page tables, stacks, data structures, etc.
  - Remember current page table is only 1! page

- Where to get memory to keep the list itself?
  - 1 level, only 4MB entries
    - You don't even have space to keep the second level page tables

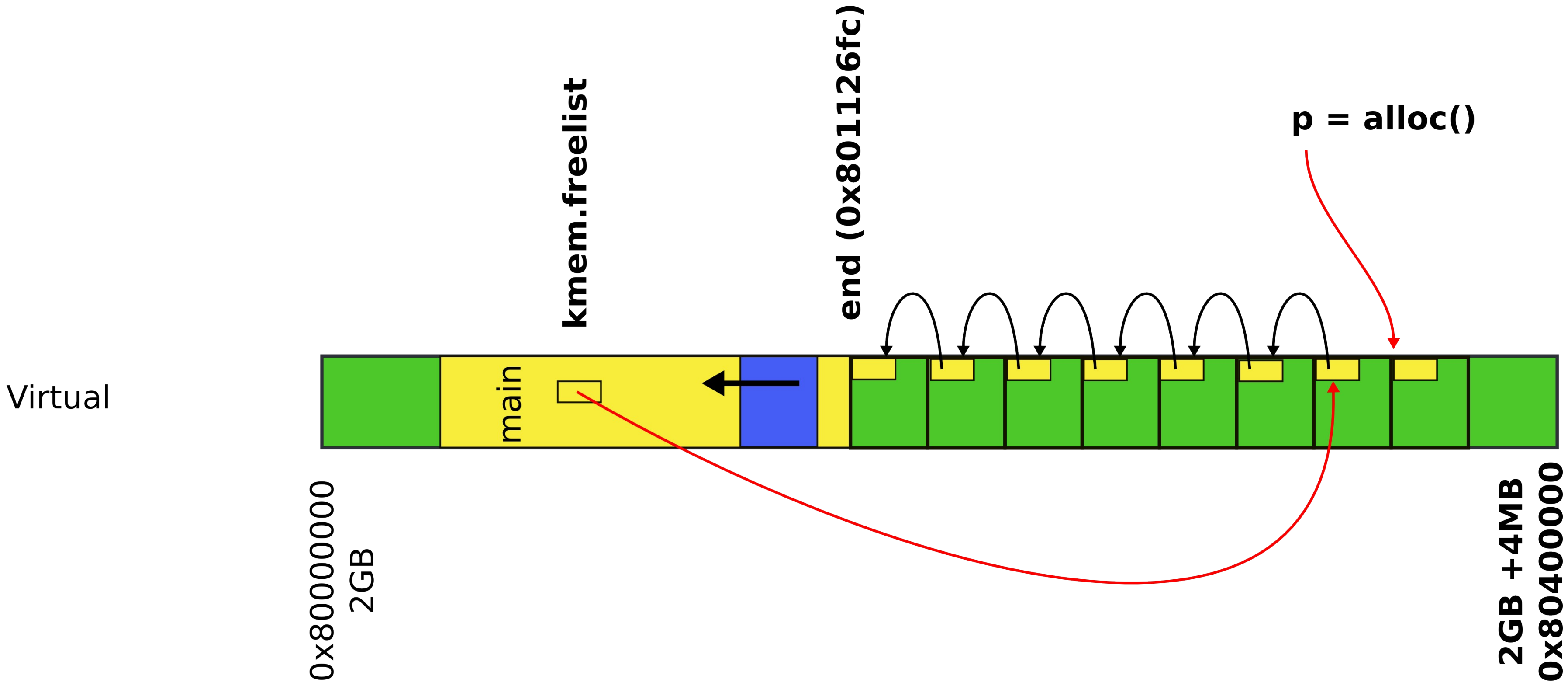
Physical page allocator

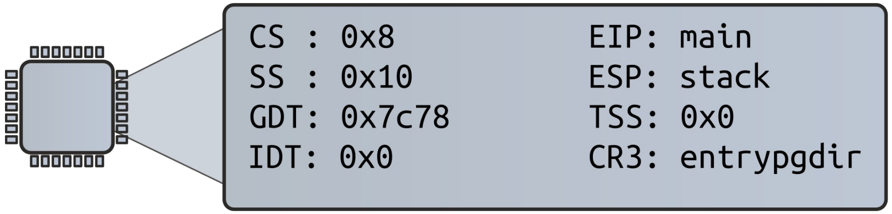


```
CS : 0x8          EIP: main
SS : 0x10         ESP: stack
GDT: 0x7c78       TSS: 0x0
IDT: 0x0          CR3: entrypgdir
```

Protected Mode

Physical page allocator

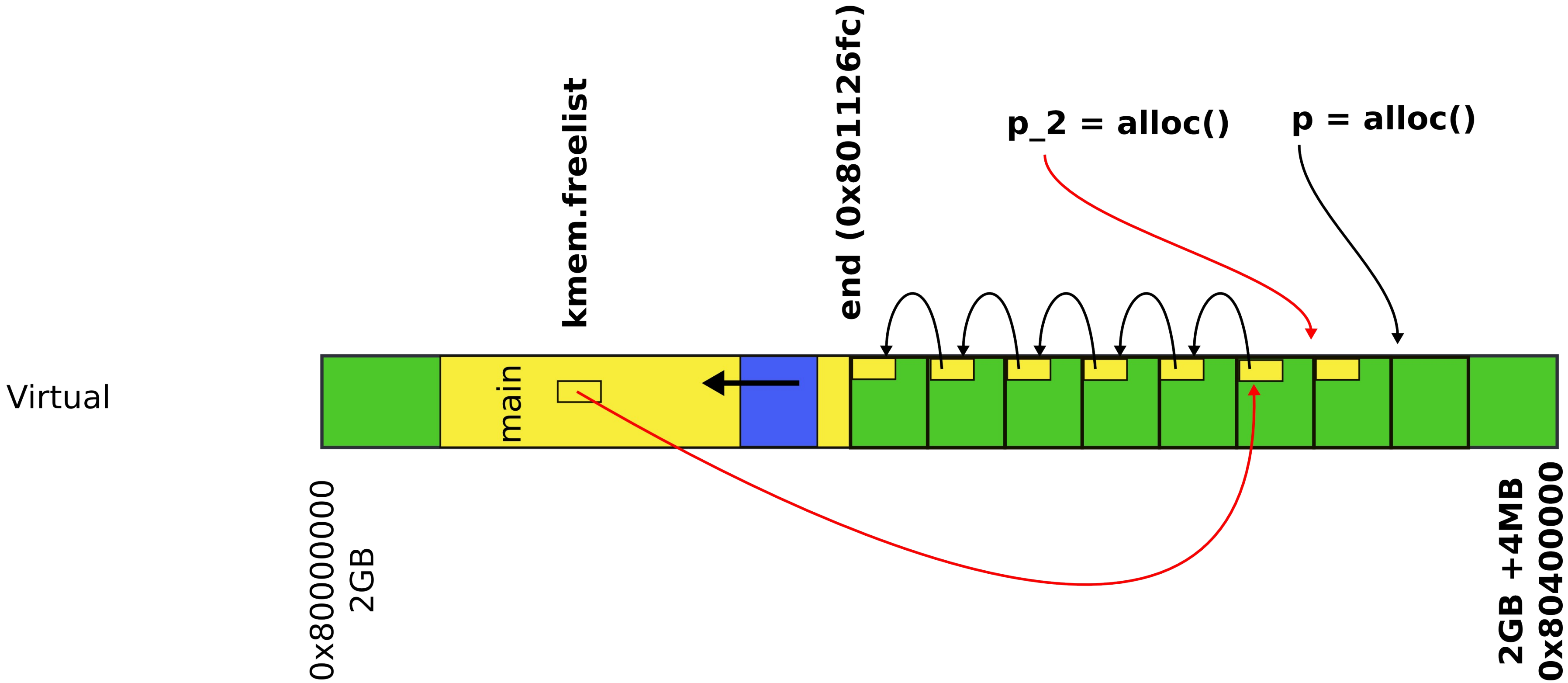


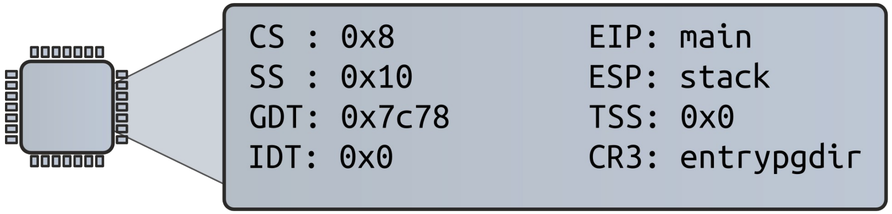


```
CS : 0x8          EIP: main
SS : 0x10         ESP: stack
GDT: 0x7c78       TSS: 0x0
IDT: 0x0          CR3: entrypgdir
```

Protected Mode

Physical page allocator

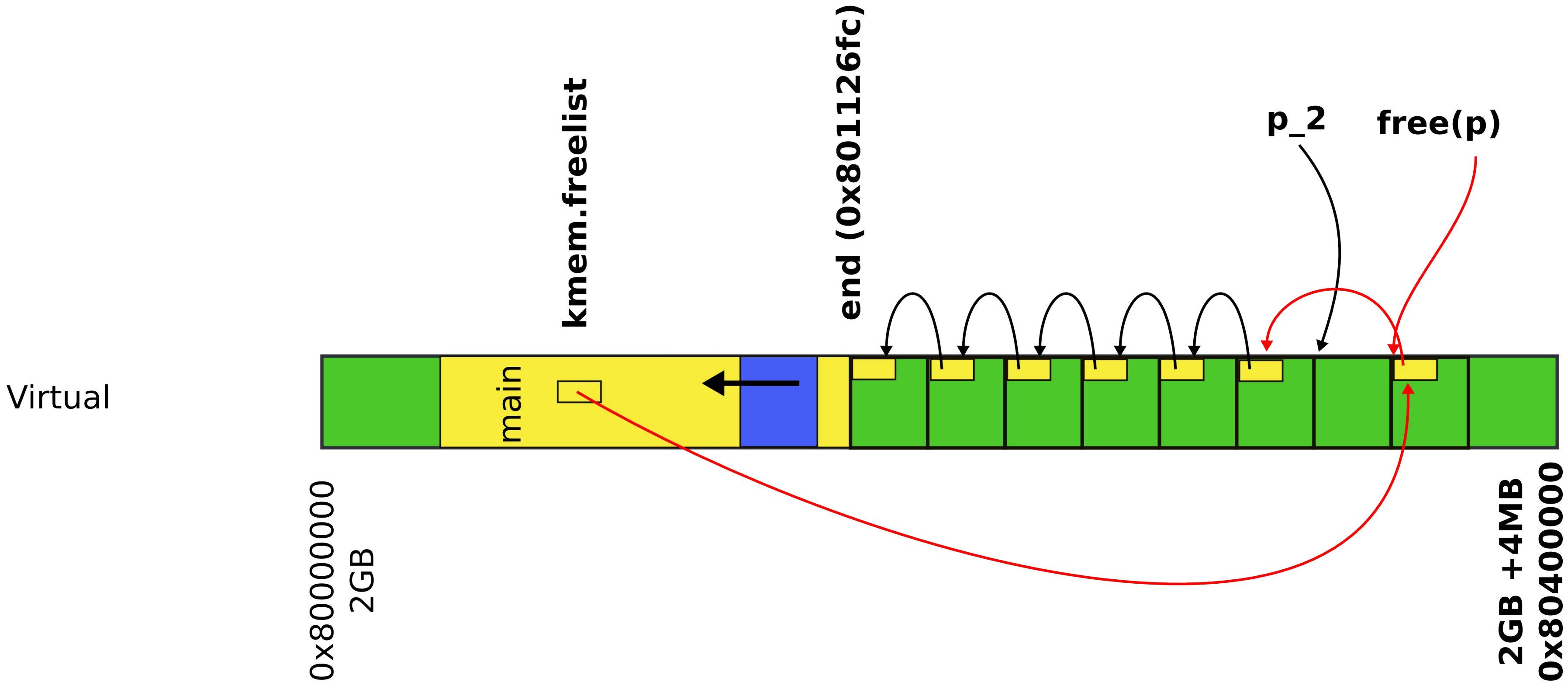




```
CS : 0x8          EIP: main
SS : 0x10         ESP: stack
GDT: 0x7c78       TSS: 0x0
IDT: 0x0          CR3: entrypgdir
```

Protected Mode

Physical page allocator



```
CS : 0x8            EIP: main
SS : 0x10           ESP: stack
GDT: 0x7c78         TSS: 0x0
IDT: 0x0            CR3: entrypgdir
```

Protected Mode

kalloc() - kernel allocator

```
char*
kalloc(void)
{
  struct run *r;
...
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
…
  return (char*)r;
}
```

```
kfree(char *v)
{
  struct run *r;
...
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
...
}
```

Kernel needs malloc()

```
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
…
}
```

Back to kinit1()

```
kinit1(void *vstart, void *vend)
{
...
  freerange(vstart, vend);
}


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

Wait! Where do we start?

```
int
main(void)
{
    kinit1(end, P2V(4*1024*1024)); // phys page allocator
    kvmalloc(); // kernel page table
    mpinit(); // detect other processors
```

- What is this **`end`**?

```
extern char end[];
```

Wait! Where do we start?

```
int
main(void)
{
    kinit1(end, P2V(4*1024*1024)); // phys page allocator
    kvmalloc(); // kernel page table
    mpinit(); // detect other processors
```

- What is this **`end`**?

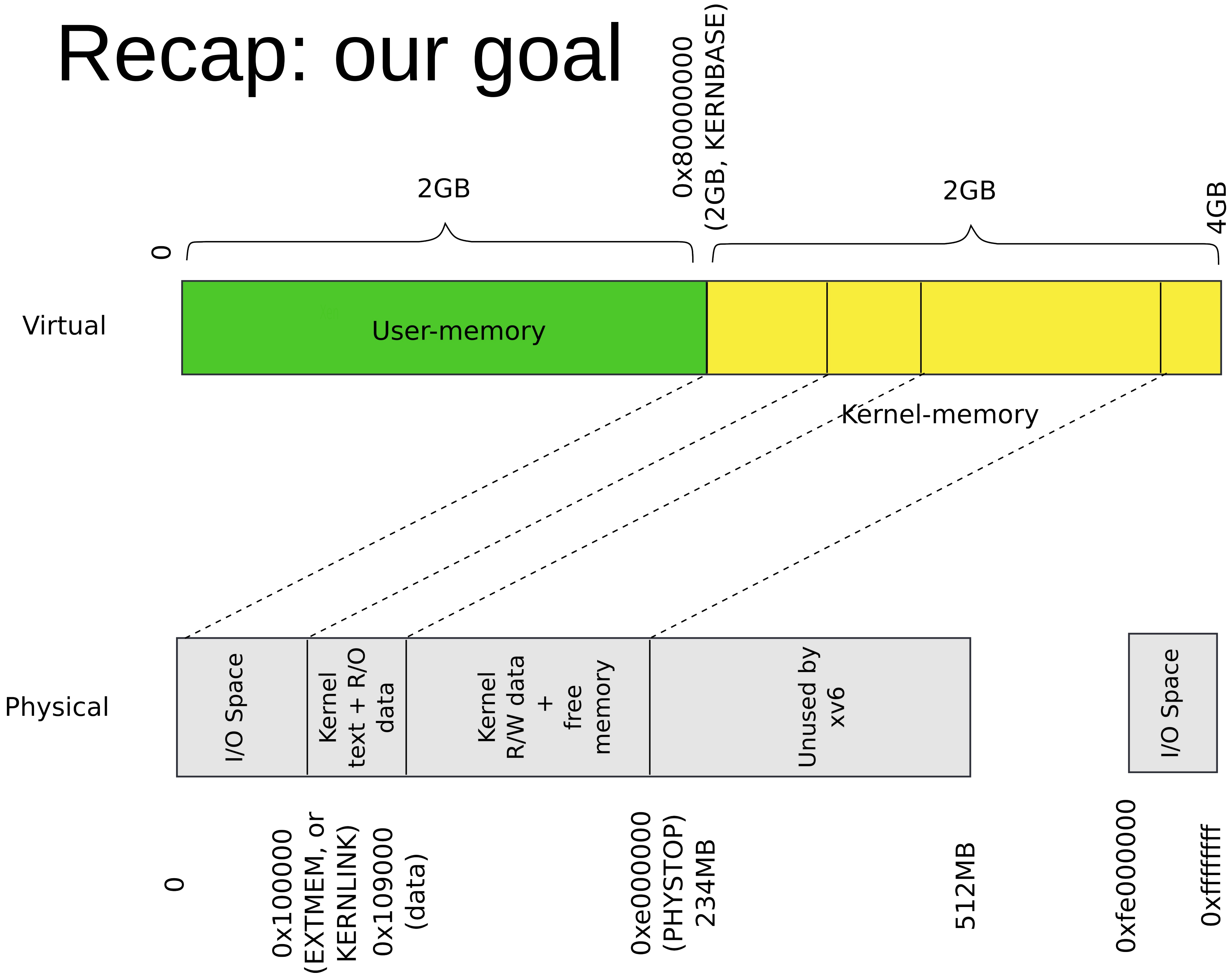
```
extern char end[]; // first address after
                      kernel loaded from ELF file
```

Back to main(): Kernel page table

```
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
…
}
```

- What do you think has to happen?
  - i.e., how to allocate page tables?

Recap: our goal



Conclusion

- Kernel has a memory allocator
  - It allocates memory in chunks of 4KB
  - Good enough to maintain kernel data structures

Thank you!