Creating kernel's page table, Initializing physical memory allocator, Running Processes in Action

Creating kernel's page table

Initializing physical memory allocator

- Remember that setupkvm allocated phys mem for page directory.

How does memory allocation work?
sheet 27

- Physical memory allocator interface:

allocates a page at a time
va = kalloc()
kfree(va)
always allocates from phys mem above where kernel was loaded
so pa is va - KERNBASE

- what does kernel use phys mem allocator for?

page table pages
kernel data structures (pipe buffers, stacks, &c)
user memory

- How does the allocator work?

data structure: a list of free physical pages
allocation = remove first entry from list
free = add page to head of list
list's "next" pointers stored in first 4 bytes of each free page
that memory is available since they are free

- Allocator depends on phys pages having virtual addresses!

since it must write them to manipulate free list
just like VM code needed to write page table pages
thus xv6 maps all phys pages into kernel address space
which burns up a lot of virtual address space, limits max user mem
other arrangements are possible

- where does allocator get initial pool of free physical memory?

kinit1
called by main
end is first address beyond the end of the kernel *as a virtual address*
memory beyond that is unused

- freerange:

PGROUNDUP since newend may not be page-aligned

- Q: why must allocated pages have page-aligned addresses?
- kinit1 assumes physical memory goes up to 4MB. Hence P2V(4*1024*1024).
- Uses this space for allocation of initial data structures (e.g., first page table kpgdir
- kinit2 assumes physical memory goes up to PHYSTOP (lame)
- kfree each page

usually called on previously-allocated memory
kinit is abusing kfree a little bit

- kfree

linked list
note cast at 2827
2828 is where we depend on phys mem being mapped at a virt addr

- kalloc takes the first element of the free list
- Q: how to allocate mem for a data structure (e.g. array) > 4096 bytes?

Processes in Action

```
process execution states:
  diagram: user/kernel, process mem, kernel thread, kern stack, pagetable
  process might be executing in user space
    with cr3 pointing to its page table
    user mode, so can use PTE_U PTEs, < 0x80000000
  or might be in a system call in the kernel
    e.g. open() finding a file on disk
    process's "kernel thread"
    kernel mode, so can use non-PTE_U PTEs
    using kernel stack
  or not currently executing

xv6 has two kinds of transitions
  trap + return: user->kernel, kernel->user
    system calls, interrupts, divide-by-zero, &c
    hw+sw saves user registers on process's kernel stack
    save user process state ... run in kernel ... restore state
  process switch: between kernel threads
    one process is waiting for input, run another
      or time-slicing between compute-bound processes
    save p1's kernel-thread state ... restore p2's kernel-thread state

Q: why per-process kernel stack?
   what would go wrong if syscall used a single global stack?

how does xv6 store process state?
  struct proc sheet 20
  kernel proc[] table has an entry for each process
  discuss pgdir, kstack, state, pid fields.
 then discuss ofile field (fd table)

discuss how any system call works.
discuss how fork system call works.
discuss how external timer interrupt works.
discuss context switching

discuss proc.tf (trapframe)
discuss proc.context (for context switch)
```