Operating Systems Engineering

xv6 & page tables

Dan Tsafrir, 2017-03-22, 2017-03-29

Reminder: goal of “processes”

- **Bring a program to life**
- **Give each process share of CPU + private memory area**
  - For code, data, stack
  - Illusion of its own dedicated machine
- **Isolation + sharing:**
  - Prevent each process from reading/writing outside its address space
  - But allow sharing when needed

HW & OS collaboration

- **OS role**
  - (De)allocate physical memory of processes
    - Create, grow, shrink remove
  - Configure HW (in our case: memory)
  - Multiplex HW (= allow for multiprogramming)
  - Keep track of processes (executing or not)
- **HW performs address translation & protection**
  - Translate user addresses to physical addresses
  - Detect & prevent accesses outside address space
  - Allow safe cross-space transfers (system calls, interrupts, …)
- **Note that**
  - OS needs its own address space
  - But should be able to easily read/write user memory

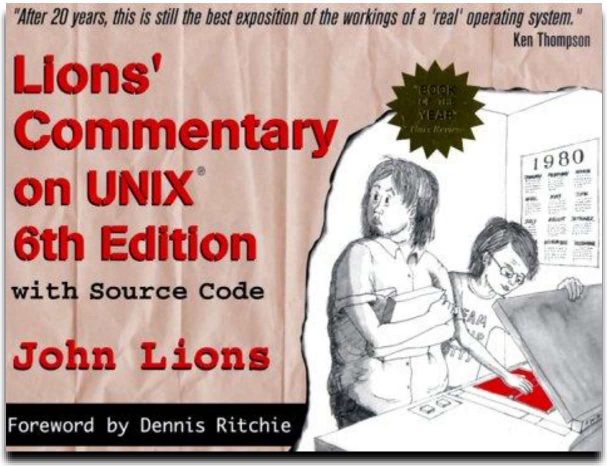
HW & OS collaboration

- **HW support not necessarily corresponds to what OS wants**
  - For example: virtual memory management in Linux/PPC (with radix tree) vs. AIX/PPC (with hash)
- **Two main approaches to x86 memory protection**
  - Segments & page tables
  - OSes typically utilize paging

Case study: Unix v6

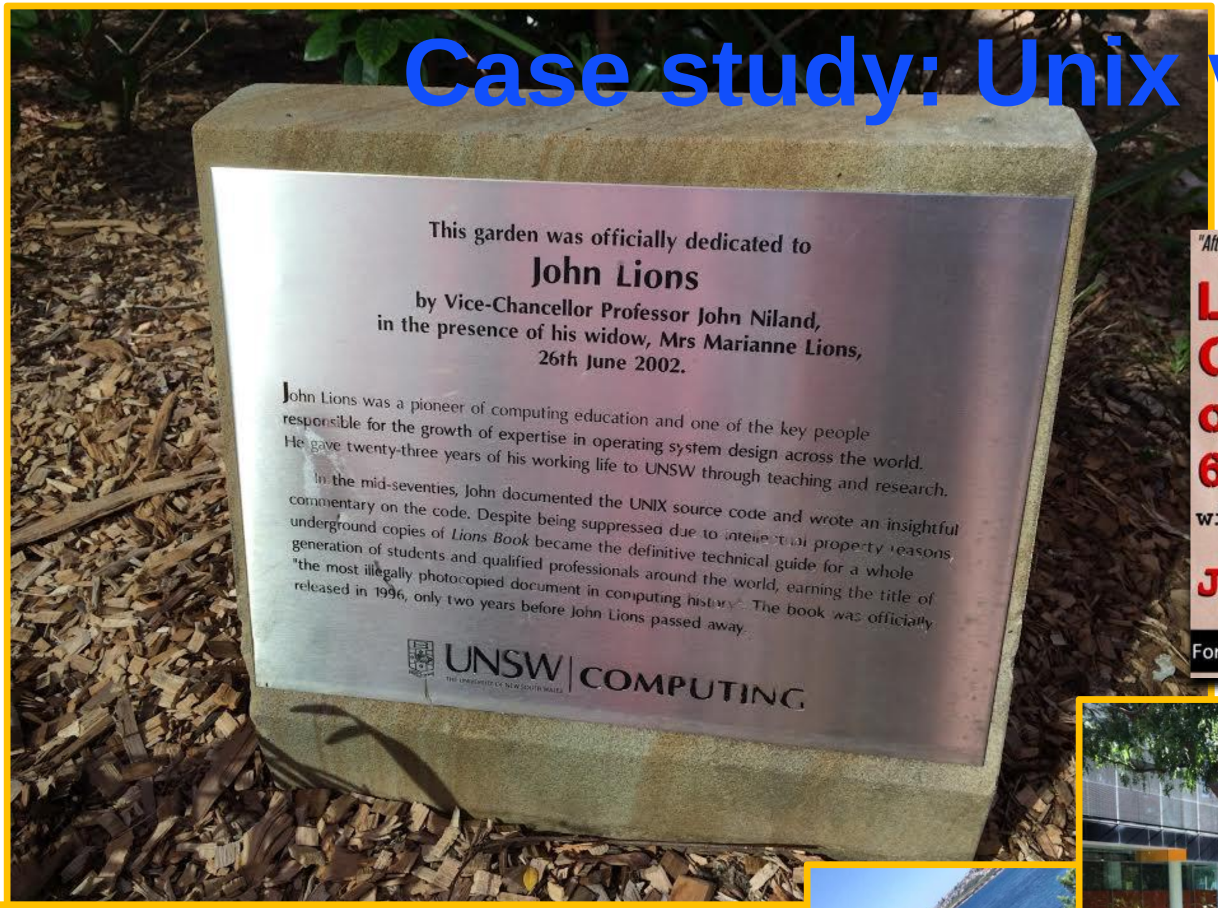
- **Early Unix OS for DEC PDP11**
  - By Ken Thompson & Dennis Ritchie, **1975**
- **From Bell labs;**
  - 1st version to be widely used outside Bell
- **Ancestor of all Unix flavors (Linux, *BSD, Solaris,…)**
  - Much smaller
- **Written in C**
  - Monolithic
  - Is recognizable even today (shell, multiuser, files, directories)
  - Today's Unix flavors have inherited many of the conceptual ideas, even though they added lots of stuff (e.g., graphics) and improved performance

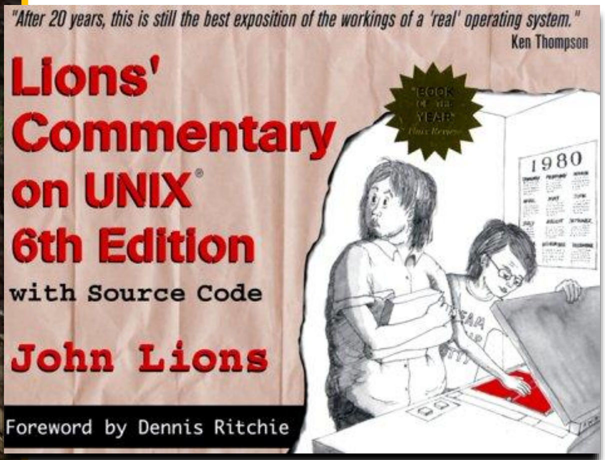
Case study: Unix v6

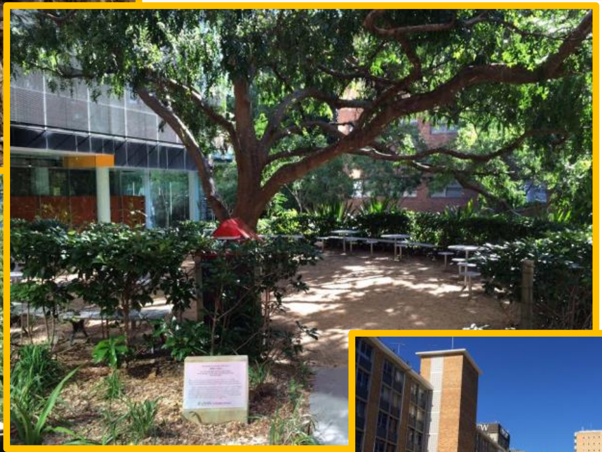


- **1976 Commentary by Lions: a classic….**
  - "Lions' Commentary on Unix 6th ed."
- **Despite its age**
  - still considered excellent commentary on simple but high quality code
- **For many years, was the only Unix kernel documentation publicly available**
  - v6 allowed classroom use of its source code; v7 & onwards didn't
- **Commonly held to be**
  - one of the most copied books in computer science
- **Reprinted in 1996** (6th edition)
  - still sold ($33.35 @ Amazon as of Mar 22, 2017)

Case study: Unix v6

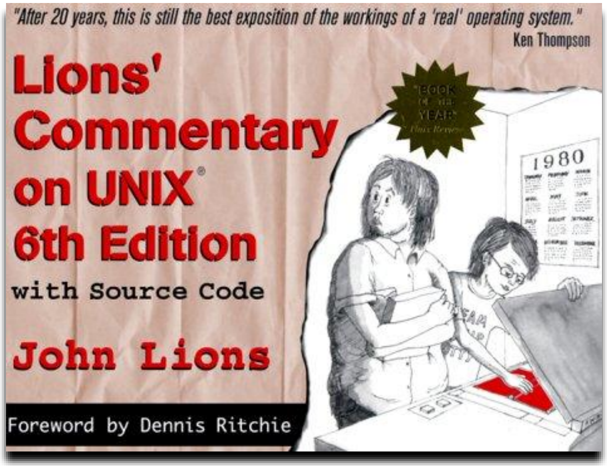






Case study: Unix v6

- **We could have used it for OSE**
  - As representing monolithic kernels
- **But**
  - PDP11
  - Old K&R C style
  - Missing some key issues in modern OSes (notably, paging, multicore)
- **Luckily…**



Case study: xv6

- **xv6 is an MIT reimplementation of Unix v6**
  - Runs on x86 if you insist
    - But we'll run it on top of QEMU
  - Even smaller than v6
  - Preserves basic structure (processes, files, pipes, etc.)
  - Runs on multicores
  - Supports paging (got paging in 2011 ☺
- **To "get it", you'll need to read its source code**
  - It's really isn't that hard
  - The xv6 commentary book (see course website) is *very* helpful

Case study: xv6

- **First half of course**
  - Lecture study source code of one xv6 part
  - Should help in homework assignments

- **About half-way the term**
  - We'll understand most of the source code for one well-designed OS for an Intel-based machine

- **Second half**
  - Covers OS concepts invented after Unix v6
  - We'll typically discuss research papers targeting these concepts
  - No tutorials from that point

xv6 – why?

- **Q: why study an aging OS?**
  - Instead of, say, Linux, or Windows, or FreeBSD, or Solaris, …
- **A1: it's big enough**
  - To illustrate basic OS design & implementation
- **A2: it's small enough**
  - To be (relatively) easily understandable
- **A3: it's similar enough**
  - To those other modern OSes
  - Once you've explored xv6, you'll find your way inside kernels such as Linux
- **A4: it'll help you**
  - To build your own (J)OS, as noted

OS *engineering*

- **JOS**
  - Occupies a very different point in the design & implementation space from xv6
- **Types of OSes**
  - Microkernel
    - QNX, L4, Minix
  - Monolithic kernel
    - xv6, Unix family (Linux, FreeBSD, NetBSD, OpenBSD, Solaris/SunOS, AIX, HPUX, IRIX, Darwin), Windows family
    - Although , actually, nowadays, most OSes are hybrid
  - Exokernel
    - “Library OS” + as few abstractions as possible
    - Many experimental systems, JOS

The 1^{st} process

- **This lecture:**
  - Chapter #2 in xv6 commentary (see course webpage)
- **(Next tutorial will do Chapter #1)**

Boot sequence

IMPLEMENTING VIRTUAL MEMORY IN XV6

xv6 address space

- **xv6 enforces memory address space isolation**
  - No process can write to another's space, or the kernel's
- **xv6 does memory "virtualization"**
  - every process's memory starts at 0 & (appears) contiguous
  - Compiler & linker expect contiguity
- **xv6 does simple page-table tricks**
  - Mapping the same memory in several address spaces (kernel's)
  - Mapping the same memory more than once in one address space (kernel's)
    - Kernel can access user pages using user virtual addresses (0…)
    - Kernel can also access user pages through kernel's "physical view" of memory
  - Guarding a user stack with an unmapped page

xv6 address space

- **x86 defines three kinds of memory addresses**
  - Virtual (used by program), which is transformed to
  - Linear (accounting for segments), which is transformed to
  - Physical (actual DRAM address)
- **xv6 nearly doesn’t use segments**
  - All their bases set to 0 (and their limits to the max)
  - virtual address = linear address
  - Henceforth we’ll use only the term “virtual”

Reminder: paging in a nutshell

- **Virtual address (we assume 32bit space & 4KB pages):**

pdx (10bit)	ptx (10bit)	page offset (12bit)

- **Given a process to run, set cr3 = pgdir ("page directory")**
- **Accessing pgdir[pdx], we find this PDE (page directory entry)**

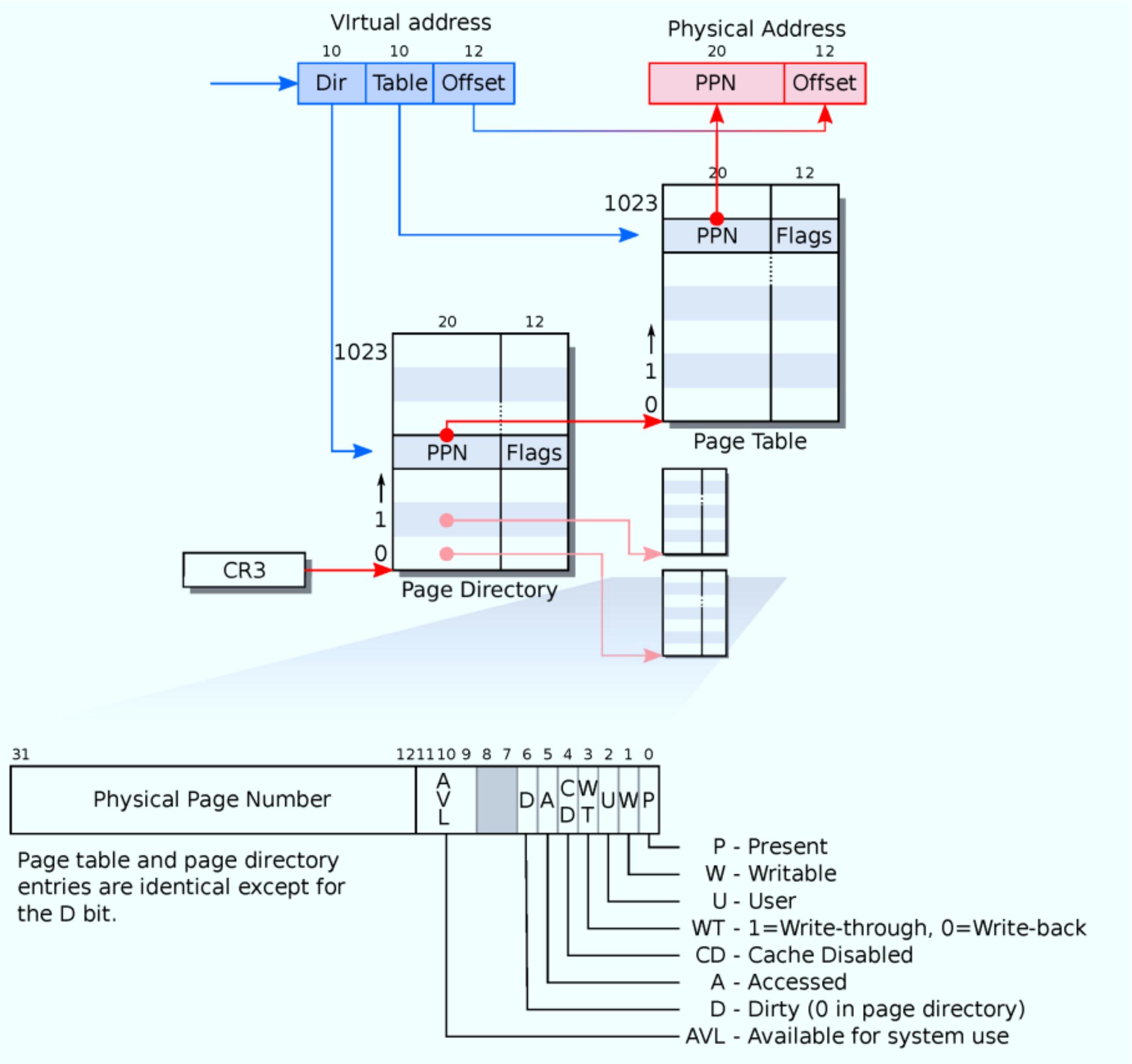
physical address (20bit)	flags (12bit)

- **Flags (bits): PTE_P (present), PTE_W (write) , PTE_U (user)**
- **20bits are enough, as we count pages of the size 2^12 (=4KB)**
- **The "page table" pgtab = pgdir[pdx] & 0x ffff f000**
  - An actual physical address
- **Accessing pgtab[ptx], we find this PTE (page table entry)**

physical address (20bit)	flags (12bit)

- **Target physical address =**
  - (pgtab[ptx] & 0x ffff f000) | $\underbrace{\text{(virt\_adrs \& 0x 0000 0fff)}}_{\text{offset}}$

paging HW



VIrtual address
10
10
12
Dir
Table
Offset
Physical Address
20
12
PPN
Offset
20
12
1023
PPN
Flags
1
0
Page Table
20
12
1023
PPN
Flags
1
0
CR3
Page Directory
31
12 11 10 9 8 7 6 5 4 3 2 1 0
Physical Page Number
A V L
D
A
C D
W T
U
W
P
Page table and page directory entries are identical except for the D bit.
P - Present
W - Writable
U - User
WT - 1=Write-through, 0=Write-back
CD - Cache Disabled
A - Accessed
D - Dirty (0 in page directory)
AVL - Available for system use

xv6 virtual memory

- **Each process has its own unique pgdir (= address space)**
- **When the process is about to run**
  - cr3 is assigned with the corresponding pgdir
- **Every process has at most KERNBASE (=2GB) memory**
  - (Actually less, since we assume PHYSTOP = 224 MB <= 2GB)
- **Kernel maps for <u>itself</u> the entire physical memory as follows:**
  - VA: *KERNBASE … KERNBASE+PHYSTOP*
    mapped to PA: *0 … PHYSTOP*
- **Such mapping exists in every v-space of *every* process**
  - PTEs corresponding to addresses higher than KERNBASE have the PTE_U bit off, so processes can't access them
- **Benefit:**
  - Kernel can use each process v-space to access physical memory
  - There exists a simple mapping from kernel v-space to *all* physical
    PA = VA - KERNBASE

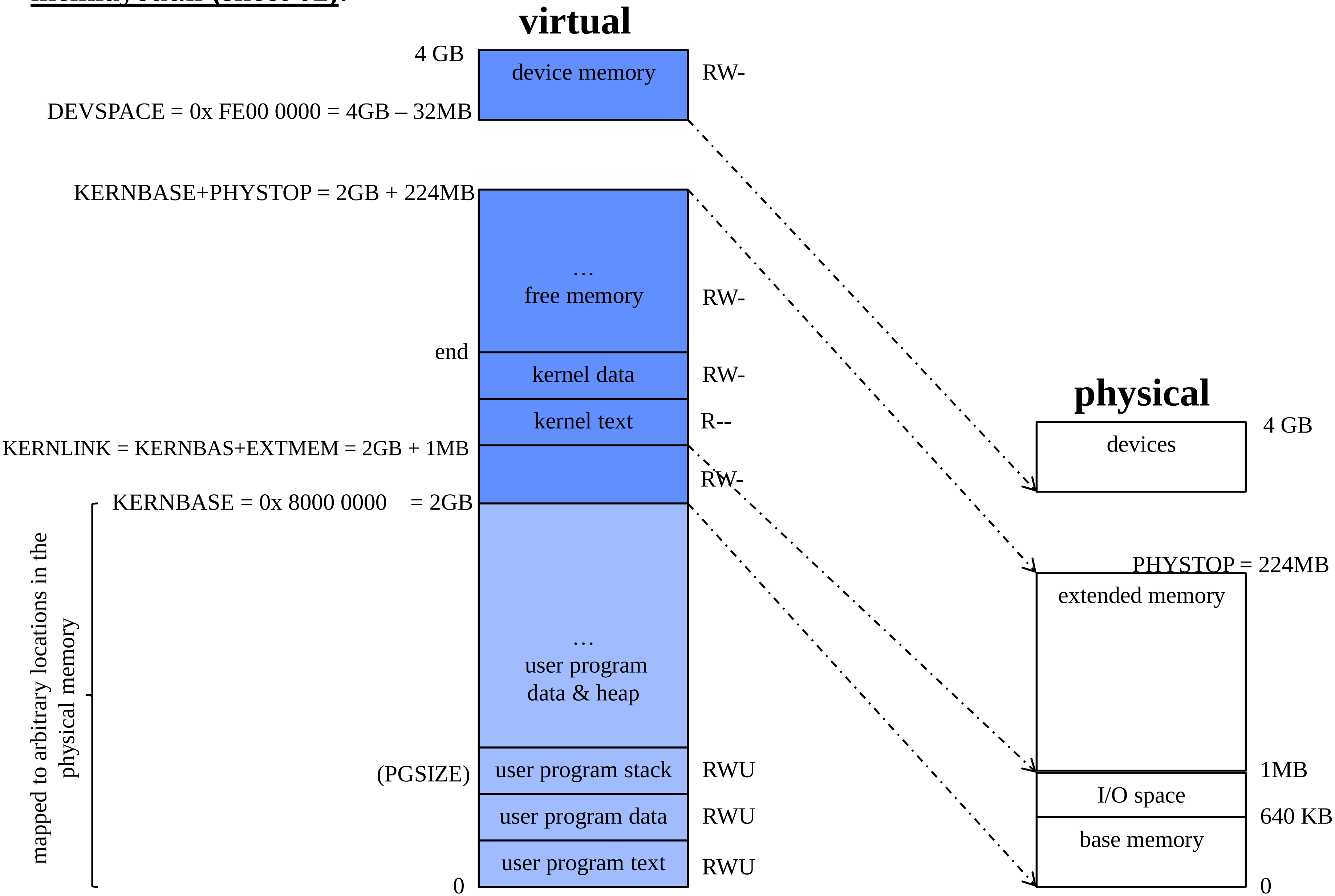
xv6 virtual memory

- **Assume**
  - Process *P* has size of 12KB (3 pages)
  - P sbrk-s (dynamically allocates) another page
- **Assume that the newly allocated *physical* page given to P is**
  - PA=0x 2010 0000
- **Thus, to ensure contiguity, the 4th PTE of *P*…**
  - (Which covers VAs: 0x 0000 3000 – 0x 0000 3fff
    as 4096 = 16^3 = 0x 0000 1000)
- **…should be mapped to**
  - 0x20100 (= the upper 20 bits of PA 0x 2010 0000)
- **So 2 different PTEs now refer to PA = 0x 2010 0000**
  - kernel: PTE assoc. w VA = KERNBASE + 0x 2010 0000, and
  - process: PTE assoc. w VA = 0x 0000 3000
- **The kernel can use both**

Code: entry page table

- **In v-space, kernel code starts at**
  - line: 0208 (memlayout.h)
  - KERNLINK = KERNBASE + EXTMEM
    = 2GB + 1MB = 0x 8010 0000
- **Why is KERNBASE (= 2GB) so high in v-space?**
  - b/c kernel v-space mapped in *each* process v-space, & want to leave enough room to allow process v-space to grow
- **Boot loader loads xv6 kernel into *physical* address 0x 0010 0000 (= 1MB)**
  - Why not at *physical* address 0x 8010 0000? (where the kernel expects to find its instructions & data in terms of VA)?
    - Because it might not exist
  - Why not at 0x0?
    - b/c first 1MB contains address ranges of legacy devices

memlayout.h (sheet 02):

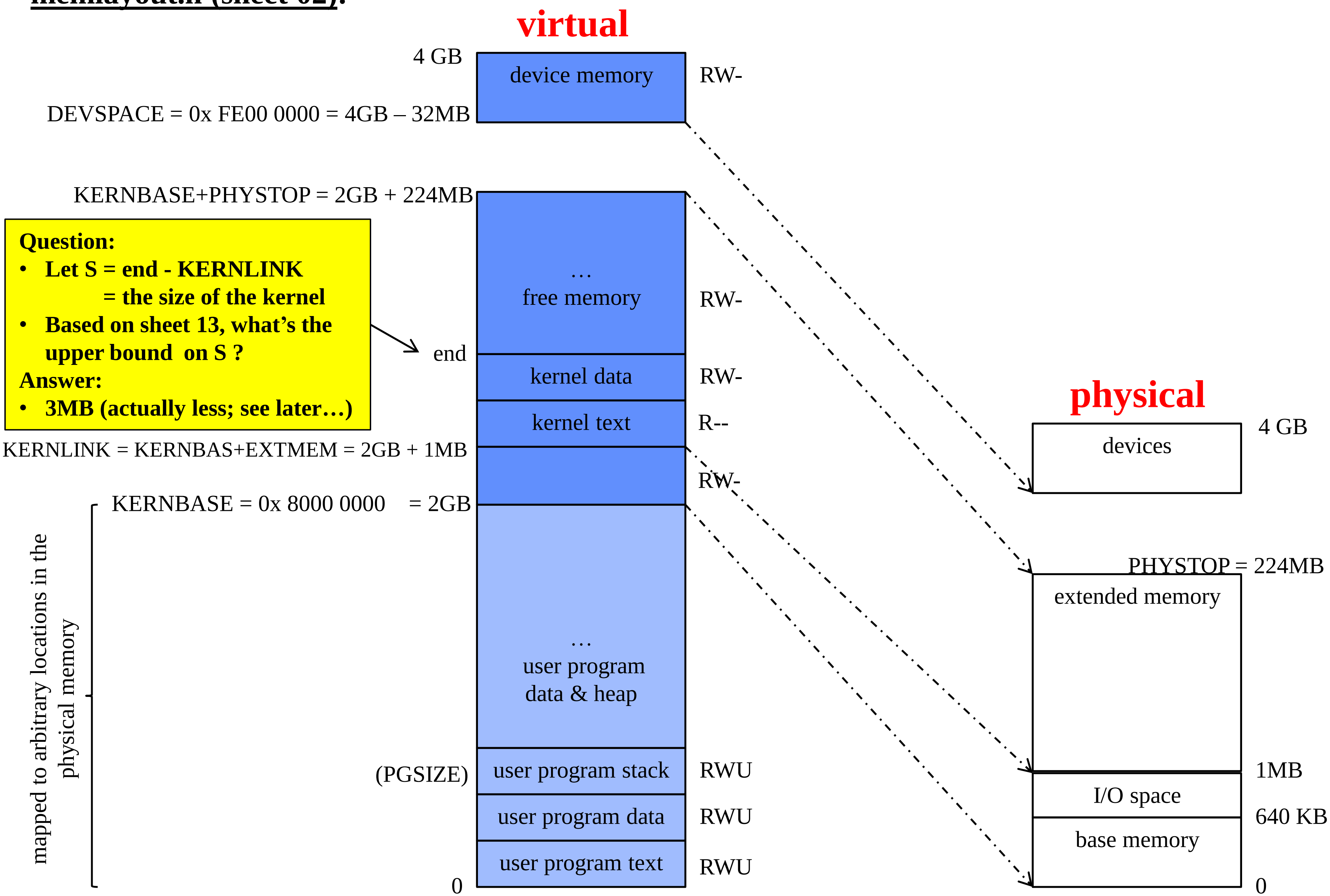


First page directory

Stopped here in previous lecture

- **PTE structure & bits (mmu.h)**
  - line: 0805
- **`entrypgdir` (main.c)**
  - The first page directory
  - line: 1310 – 1316 (after understanding, see next slide)
  - Used by...
- **`entry` (entry.S)**
  - The boot-loader loads xv6 from disk & starts executing here
  - line: 1040 … 1061
    (V2P_WO defined at 0220 – sheet 2)

memlayout.h (sheet 02):



Question:
- **Let S = end - KERNLINK**
  = the size of the kernel
- **Based on sheet 13, what's the upper bound on S ?**

Answer:
- **3MB (actually less; see later…)**

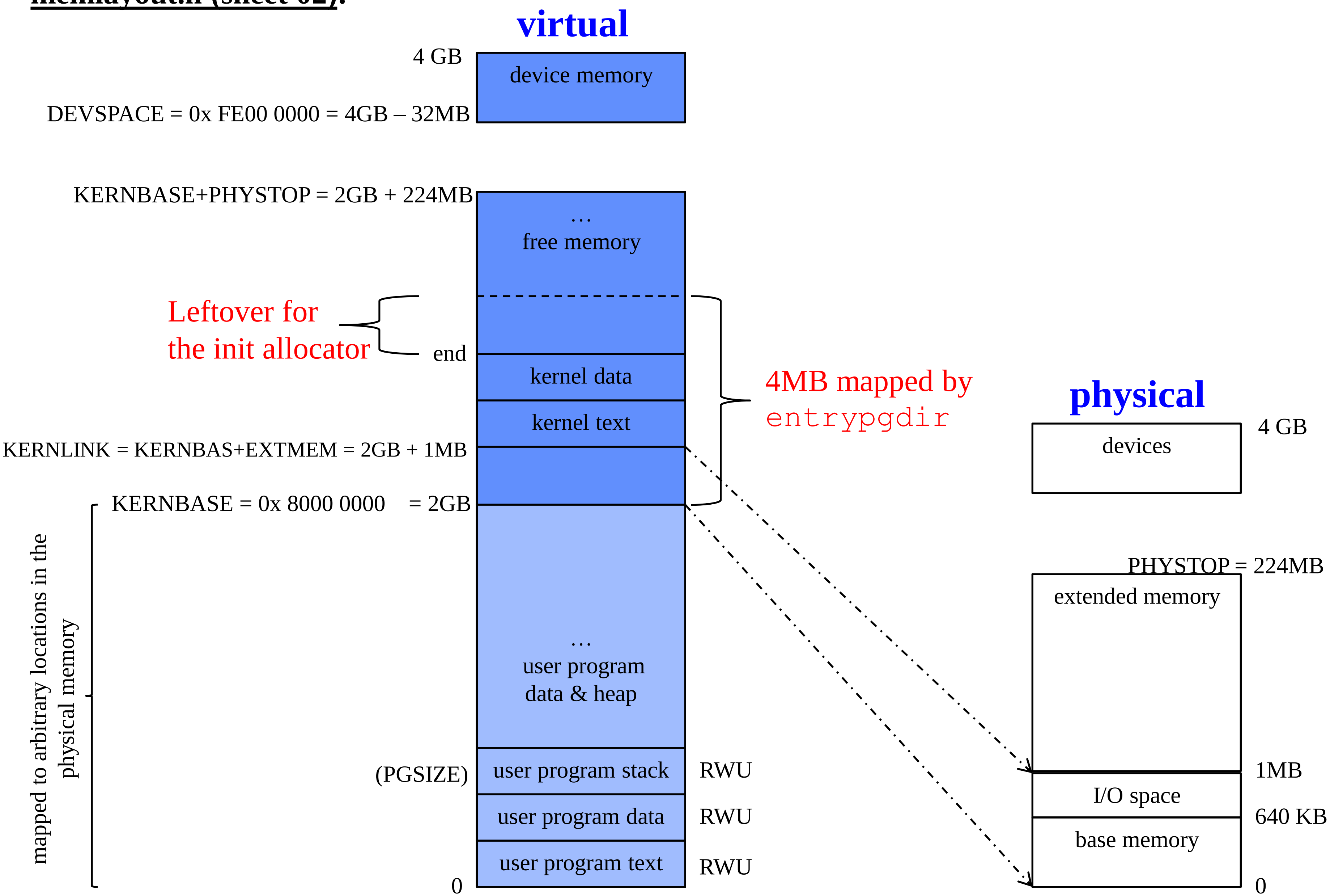
Creating an address space

- **`main`**
  - line: 1217
- **2nd line of main: `kvmalloc` (we'll get back to the 1st)**
  - in vm.c (implementation of virtual memory)
  - line: 1757; switch to page table that maps all memory
  - `kmap` array (1728) + `setupkvm` (1737)
    => `mappages` (1679)
    => `walpgdir` (1654)

Physical memory allocator

- **Maintain a free-list of all free 4KB-pages in system**
  - From end of kernel to PHYSTOP
- **Bootstrap problem**
  - Entire physical memory must be mapped in order for the allocator to initialize the free list
  - But creating a page table with those mappings involves allocating a hierarchy of page-table pages
  - xv6 solves this problem by using a separate page allocator during entry, which allocates memory just after the end of the kernel's data segment
  - This allocator does not support freeing & is limited by 4 MB mapping in the `entrypgdir`, but our kernel is small enough for it to be sufficient to allocate the first kernel page table

memlayout.h (sheet 02):



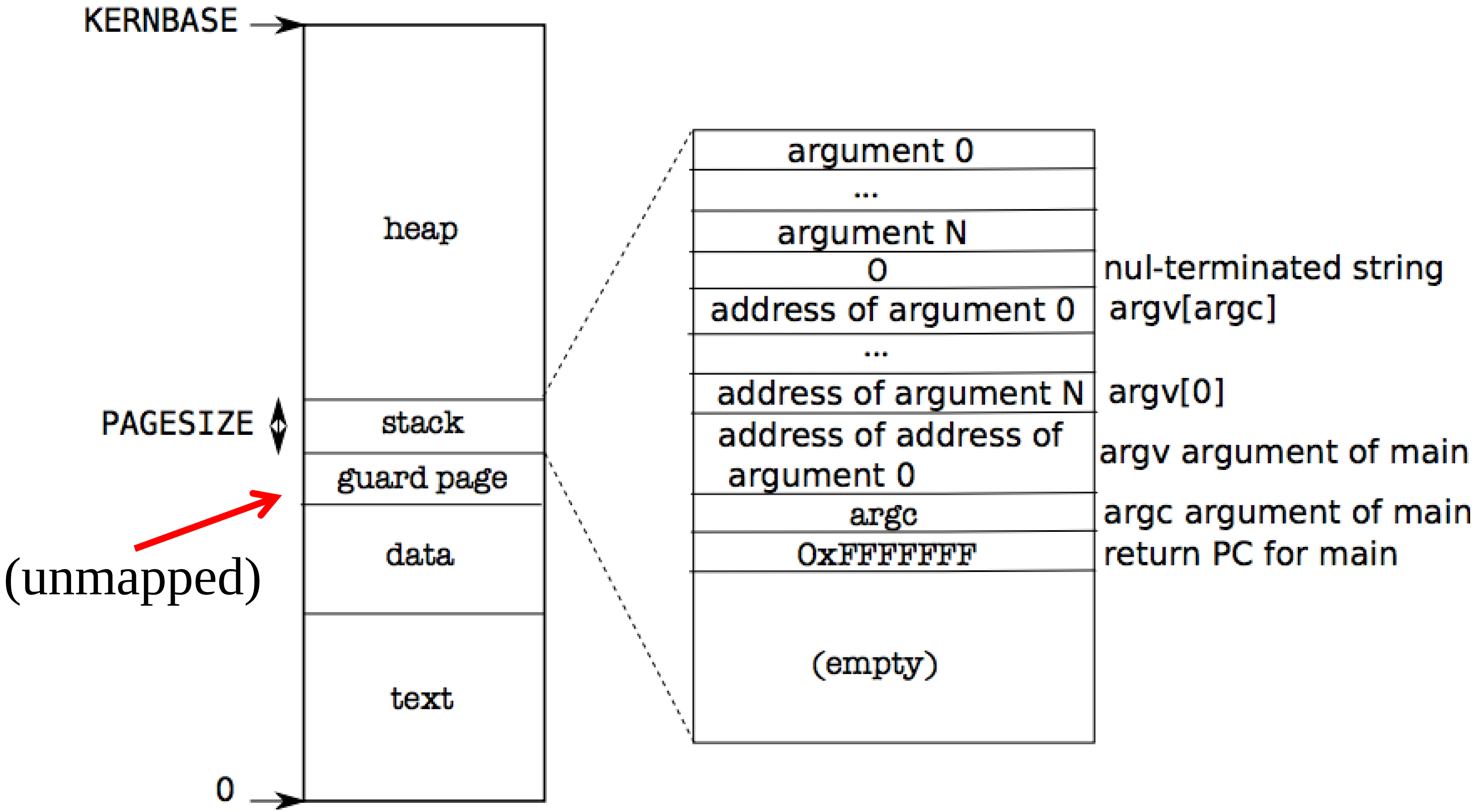
Physical memory allocator

- **The 2 free lists (for init phase, and for normal runs)**
  - File: kalloc.c = the kernel's allocator
  - Representation: `struct run` (2764) and `kmem` (2772)
    - '`next`' saved in chained pages
  - `kinit1` (2780), `kinit2` (2788)
    - Called from `main` (rows: 1219, 1238)
  - `freerange` (2801), `kfree` (2815)
  - Allocation of a page: `kalloc` (2838)

Homework

- Read pages 30 – 32 in xv6 commentary

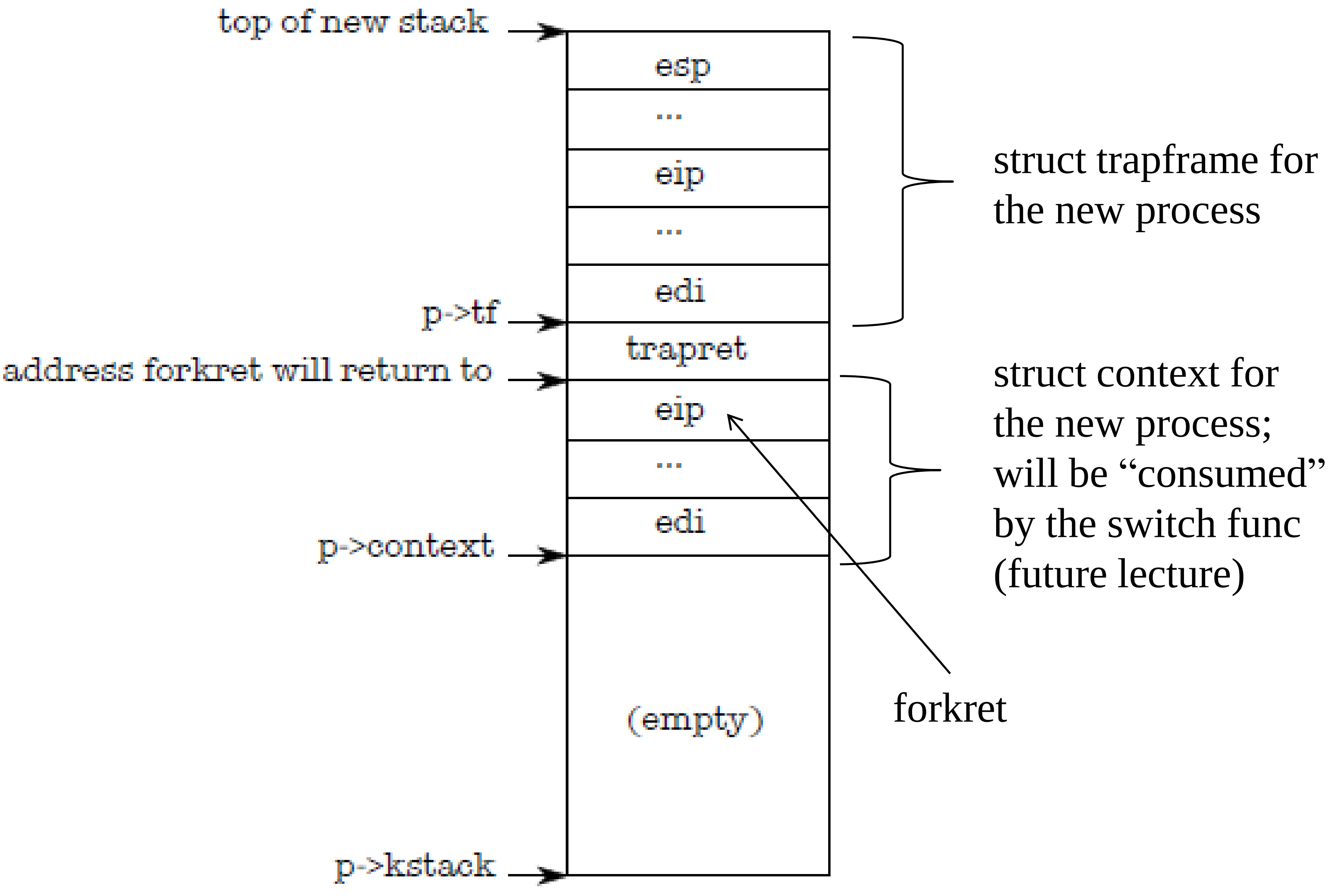
User part of an address space



END

Process creation

- **struct proc (lines: 2053-2067)**
  - Why do we need a kernel stack for every process?
  - Why can't the kernel use the user stack?
- **main (line 1237) => userinit (line 2202) => allocproc (line 2155)**
  - userinit creates the first process (init) using allocproc
  - allocproc creates all processes (used by fork
- **allocproc (lines: 2155-2194)**
  - find empty proc entry
  - allcate pid and set state
  - create the kernel stack of the new process as in the next slide
- **userinit (lines: 2202-2226)**
  - inituvn (1786), initcode.S (7500)


top of new stack
esp
...
eip
...
edi
p->tf
trapret
address forkret will return to
eip
...
edi
p->context
(empty)
p->kstack
struct trapframe for the new process
struct context for the new process; will be "consumed" by the switch func (future lecture)
forkret

Running a process

- main => userinit => mpmain (1267) => scheduler (2408) => switchuvm (1764)