# CS 134
# Operating Systems

Feb 13, 2019

## Virtual Memory, 2 of 2

# Segmentation vs. Paging

| | Segmentation | Paging |
|---|---|---|
| Need the programmer be aware the technique is being used? | | |
| How many linear address spaces are there? | | |
| Can the total address space exceed the size of phys. mem? | | |
| Can procedures and data be distinguished and separately protected? | | |
| Can tables whose size fluctuates be accommodated easily? | | |
| Is sharing of procedures between users facilitated? | | |

# 101 fun things to do with Paging Hardware

- Better performance/efficiency
  - Demand sbrk allocation
  - Demand stack allocation
  - One zero-filled page
  - Copy-on-write fork
  - Demand Paging
- New features
  - Memory-mapped files
  - Shared memory
  - Virtual Memory

# On-demand page allocation

- `sbrk` is the system call to allocate more memory for a process.
  - Difficult for applications to predict in advance
  - sbrk allocates memory that may never be used
- Allocate lazily
  - When sbrk call is made, allocate address space for new memory (but not physical pages).
  - As logical pages are accessed, insert physical pages for them.

# x86 page faults

- One of the few dozen exceptions on x86 is T_PGFLT
- Generates controlled transfer into the kernel (like a trap)
- Information needed to handle the fault:
  - The virtual address that caused the fault
  - The type of violation that caused the fault (e.g., RW)
  - The EIP and CPL when the fault occurred

```
// Layout of the trap frame built on the stack by the
// hardware and by trapasm.S, and passed to trap().
struct trapframe {
  // registers as pushed by pusha
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame
  ushort gs;
  ushort padding1;
  ushort fs;
  ushort padding2;
  ushort es;
  ushort padding3;
  ushort ds;
  ushort padding4;
  uint trapno;

  // below here defined by x86 hardware
  uint err;
  uint eip;
  ushort cs;
  ushort padding5;
  uint eflags;

  // below here only when crossing rings, such as from user to kernel
  uint esp;
  ushort ss;
  ushort padding6;
};
```

Pushed by SW trap handler

Type of fault

More detailed reason for fault

Pushed by x86 HX

x86.h

# Dispatching traps

- x86 references a special table called the *interrupt descriptor table* (IDT)
- IDT is an array of function handlers for each possible exception
- Some exceptions, like page faults, push additional error codes on the stack (others don't)
- For all exceptions/interrupts, HW pushes EIP, CS, CFLAGS, etc.

# Handling exceptions

```
…
.globl vector11
vector11:
  pushl $11
  jmp alltraps
.globl vector12
vector12:
  pushl $12
  jmp alltraps
.globl vector13
vector13:
  pushl $13
  jmp alltraps
.globl vector14
vector14:
  pushl $14
  jmp alltraps
.globl vector15
…
```

vectors.S

T_PGFLT

One vector handler in the IDT for each possible exception

vectors.S is generated by vectors.pl

# Handling exceptions

```
#include "mmu.h"
  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal
  # Set up data segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es
  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp
  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

Construct SW portion of trapframe

Enter Kernel C Code

trapasm.S

# Gathering info to handle a page fault

- The VA that caused the fault
  - `movl %cr2, %eax` (or `rcr2()` in xv6)
- The type of violation that caused the fault
  - `tf->err` contains flag bits
    - FEC_PR: page fault caused by protection violation
    - FEC_WR: page fault caused by write
    - FEC_U: page fault caused in user mode
- The EIP and CPL where the fault occurred
  - EIP: `tf->eip`
  - CPL: `tf->cs & 0x3 > 0`
    - or check for `(tf->err & FEC_U) > 0)`

# Homework 4 solution: changes to `sys_sbrk`

```
int
sys_sbrk(void)
{
  int addr;
  int n;

  if(argint(0, &n) < 0)
    return -1;
  addr = myproc()->sz;
#if 0
  if(growproc(n) < 0)
    return -1;
#else
  myproc()->sz += n;
#endif
  return addr;
}
```

sysproc.c

Don’t allocate physical memory;
just update `myproc()->sz`

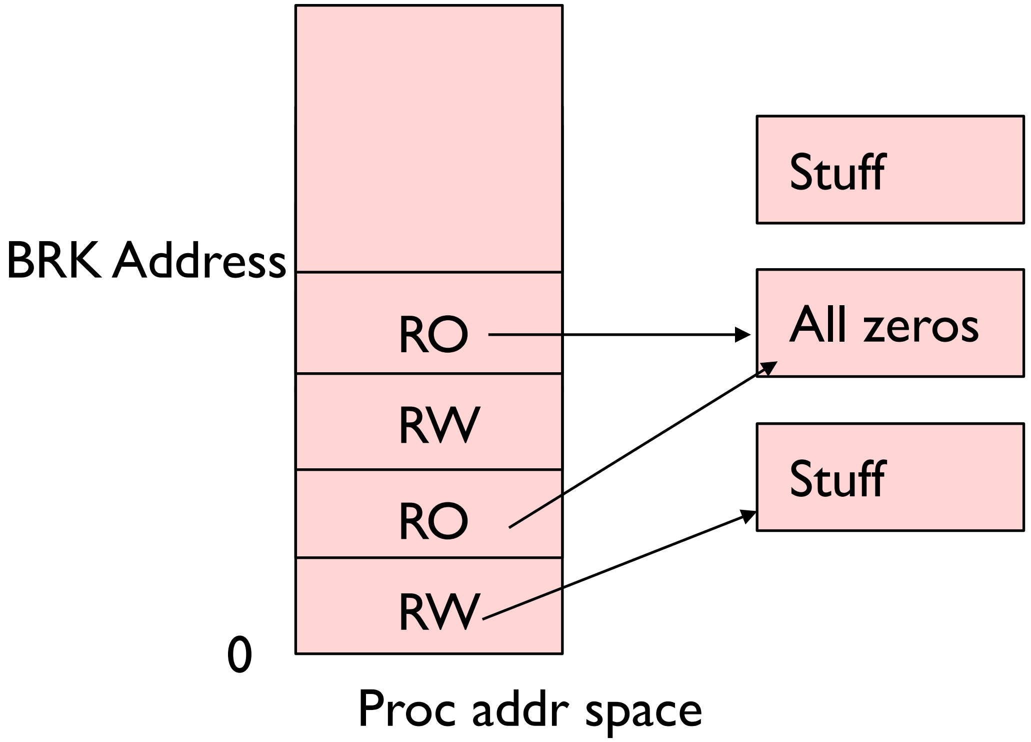
# Homework 4 solution: changes to `trap`

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    …
 }

  if (tf->trapno == T_PGFLT) {
    uint addr = PGROUNDDOWN(rcr2());
    if (addr < myproc()->sz) {
      char *mem = kalloc();

      if (!mem) {
        cprintf("out of memory");
        exit();
        return;
      }
      memset(mem, '\0', PGSIZE);
      cprintf("kernel faulting in page at %x\n", addr);
      mappages(myproc()->pgdir, (void *) addr, PGSIZE, V2P(mem), PTE_W|PTE_U);
      return;
    }
  }
  …
}
```

trap.c

# Optimization: one zero-filled page

- Observation: some sbrk'ed memory is never written to
- All sbrk'ed memory gets initialized to zero.
- Idea: Use just **one** zero page for all sbrk'ed memory.
- Copy the zero page on write (COW)



# Zero page support: changes to `trap`

```c
void
trap(struct trapframe *tf)
{
  …
  if (tf->trapno == T_PGFLT) {
    uint addr = PGROUNDDOWN(rcr2());
    int write = (tf->err & FEC_WR) > 0;
    if (addr < myproc()->sz) {
      if (write) {
        char *mem = kalloc();

        if (!mem) {
          cprintf("out of memory");
          exit();
          return;
        }
        memset(mem, '\0', PGSIZE);
        cprintf("kernel faulting in read/write page at %x\n", addr);
        mappages(myproc()->pgdir, (void *) addr, PGSIZE, V2P(mem), PTE_W|PTE_U);
      } else {
        cprintf("kernel faulting in read-only page at %x\n", addr);
        mappages(myproc()->pgdir, (void *) addr, PGSIZE, V2P(zeropg), PTE_U);
      }
      return;
    }
  }
  …
}
```

trap.c

# Optimization: dynamic stack

- Rather than allocate enough physical pages for max size of stack, allocate one to begin with
- If more stack space is used, page fault will be generated:
  - Allocate the stack space at that point.

Will the page fault necessarily be one page before the top of stack?

# Optimization: copy-on-write fork

- `fork` copies all pages in the process
- But, often, `exec` is called immediately after the `fork`
  - Which will free the newly-copied pages
- Idea: modify `fork` to mark pages copy-on-write
  - All pages in both processes become read-only
  - On page fault, copy page and mark R/W
  - Extra PTE bits (AVL) useful for indicating COW mappings)

# Optimization: Demand paging

- Observation: `exec` loads entire executable into physical memory
- But, often, not all pages of the executable are used
  - Slower exec
  - Wasted physical pages
- Idea: modify `exec` to mark code pages not present in PTEs
  - On page fault, read corresponding disk block (from executable) and install PTE
- Challenges:
  - What if file is larger than physical memory?
  - What if executable is deleted while it is running?

# Virtual memory: exceed physical memory

- Idea: Use fast (small, expensive) memory as a cache for slow (large, cheap) disk
  - 90/10 rule: processes spend 90% of their time in 10% of the code
  - Not all of a process's address space needs to be in memory at one time
  - Illusion of near-infinite memory
  - More processes in memory (higher degree of multiprogramming)
- Locality:
  - Spatial: The likelihood of accessing a resource is higher if a resource close to it was just referenced
  - Temporal: The likelihood of accessing a resource

# VM Page fault handler

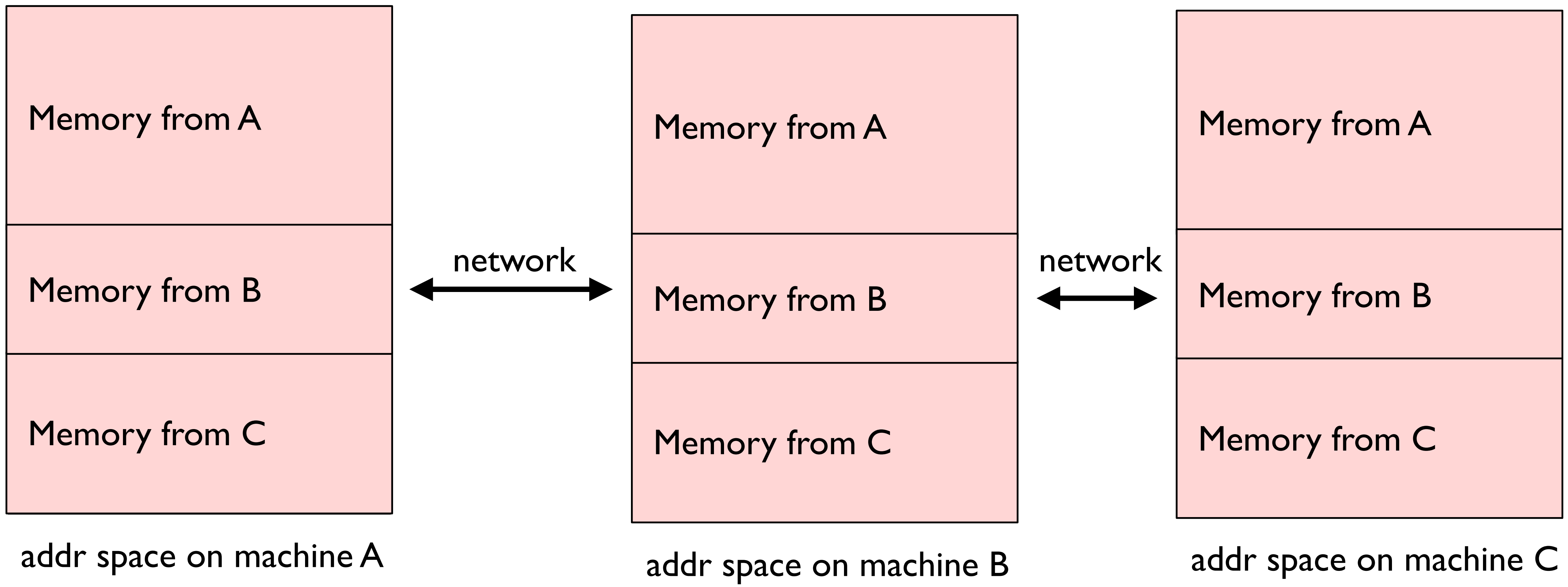
- Save registers
- Figure out virtual address that caused the fault
- If protection problem, signal or kill process
- If no free page, evict a page from memory
  - If modified, write to backing store
  - Keep disk location of this page (not in page table)
  - Suspend faulting process (resume when write complete)
- Read data from backing store for faulting page
  - From backing store or executable or fill-with-zero
  - Suspend faulting process (resume when read complete)
  - Update page table
  - Restart instruction

# Feature: memory-mapped files

- Normally, files are accessed through open/close/read/write/seek
- Idea: map file into address space
  - New system call mmap() can place file at a location in user address space
  - Kernel must read/write to the file, similar to the way the page fault handler pages in from an executable
- Processes can read/write using memory accesses rather than file read/write
  - Written data is cached in page frame
  - Difficult to change EOF of the file
- Can be shared between processes

# Feature: distributed shared memory

- Idea: use virtual memory to pretend physical memory is shared between machines



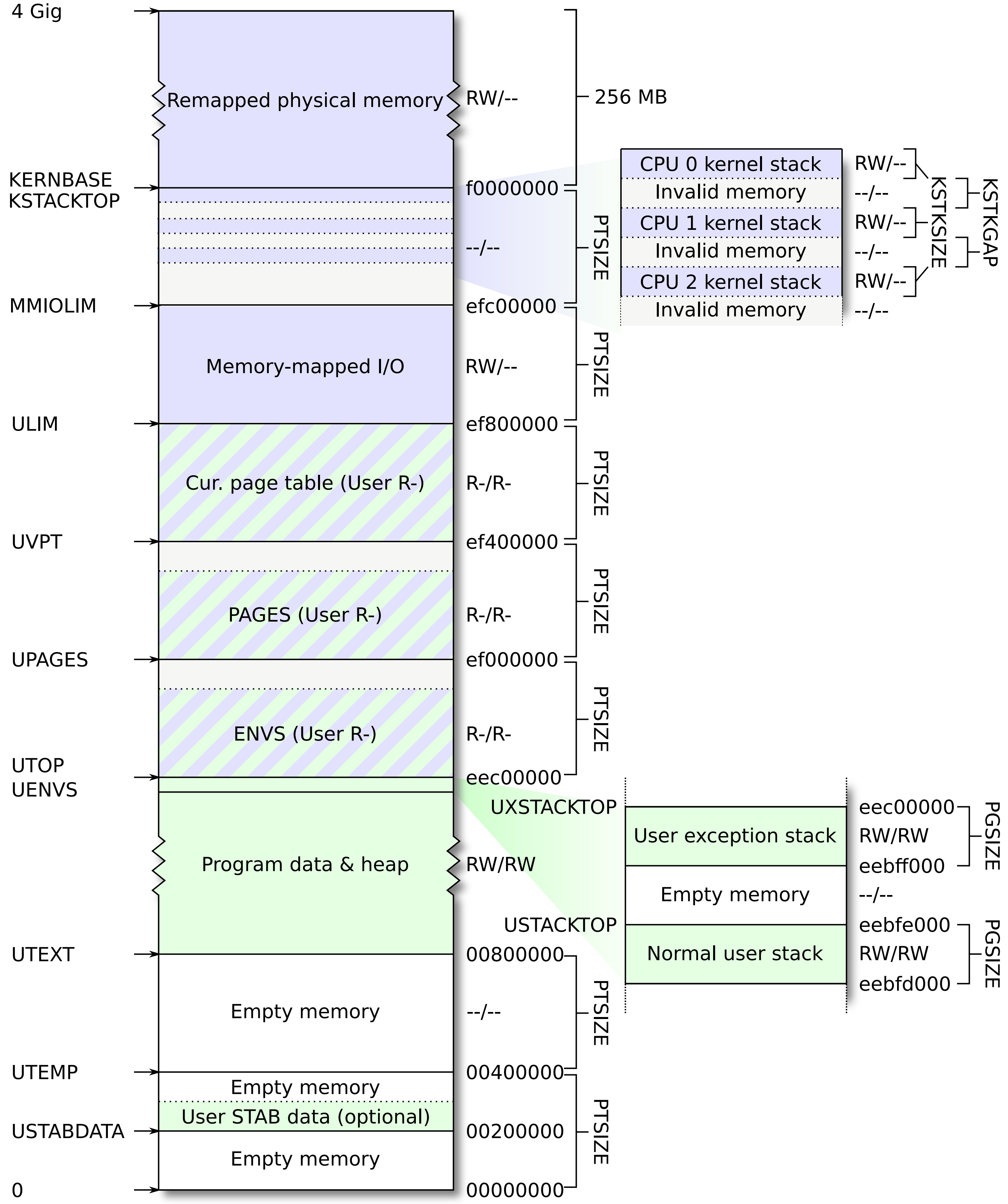
# Page Sizes

- Advantages of smaller page size
  - Less internal fragmentation
- Advantages of larger page sizes
  - TLB covers more bytes, so TLB hit rate is higher
  - Smaller page
- Page sizes have tended to increase over time
  - 1970s: Vax: 512-byte pages
  - 1980s: x86: 4 KiB
  - 1990s: Pentium (x86) 4Kib (or 4MiB)
  - 2010s: Risc V: 4KiB or 4MiB or 1 GiB or 512GiB

# Thrashing

- What it is:
  - Spending more time paging than doing real work
- Why it happens:
  - If the degree of multiprogramming is too big, each process's working set is not resident
- Solution:
  - Reduce degree of multiprogramming. Swap entire processes out to disk

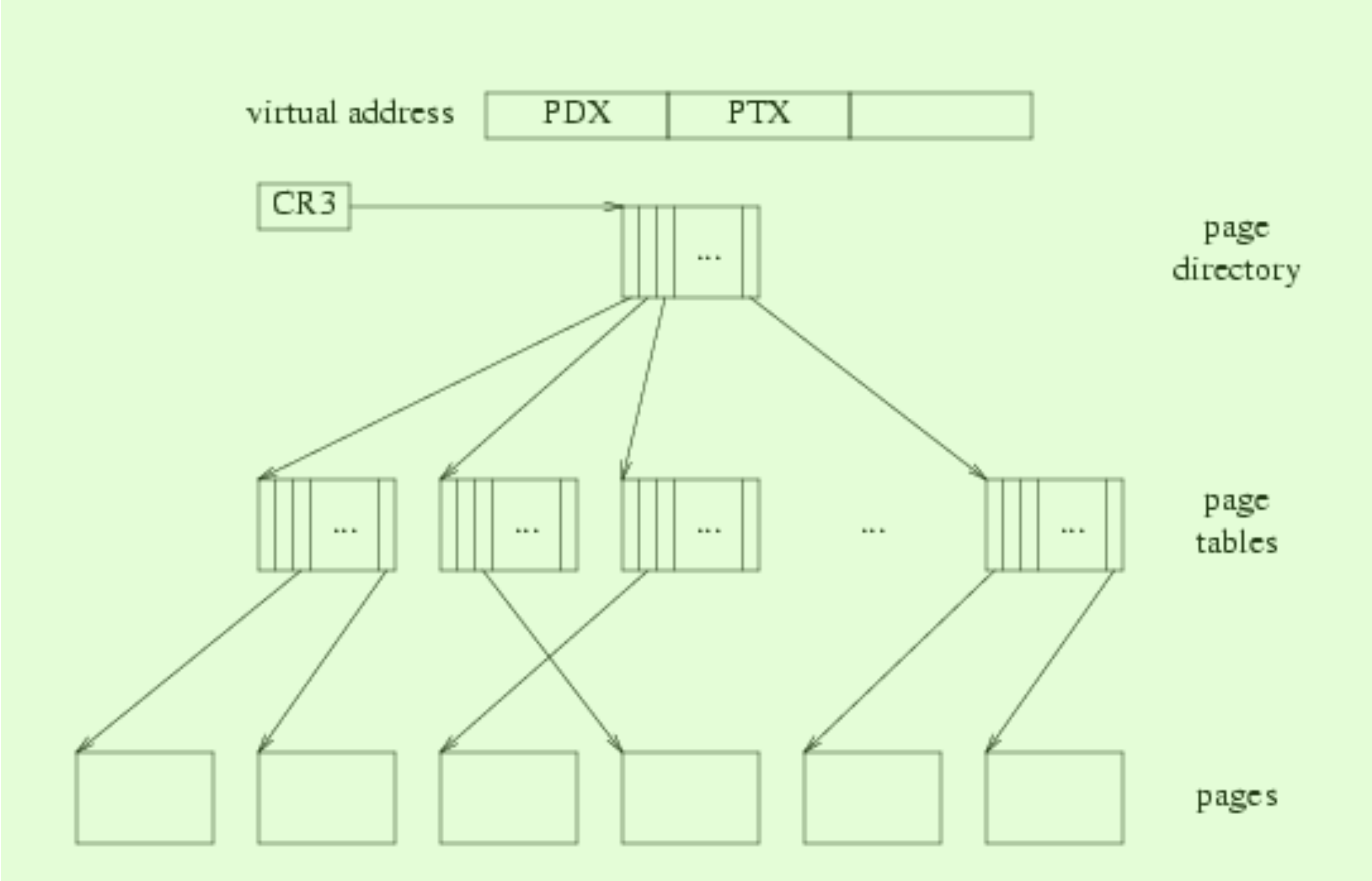
# Jos Virtual Memory layout



# JOS UVPT

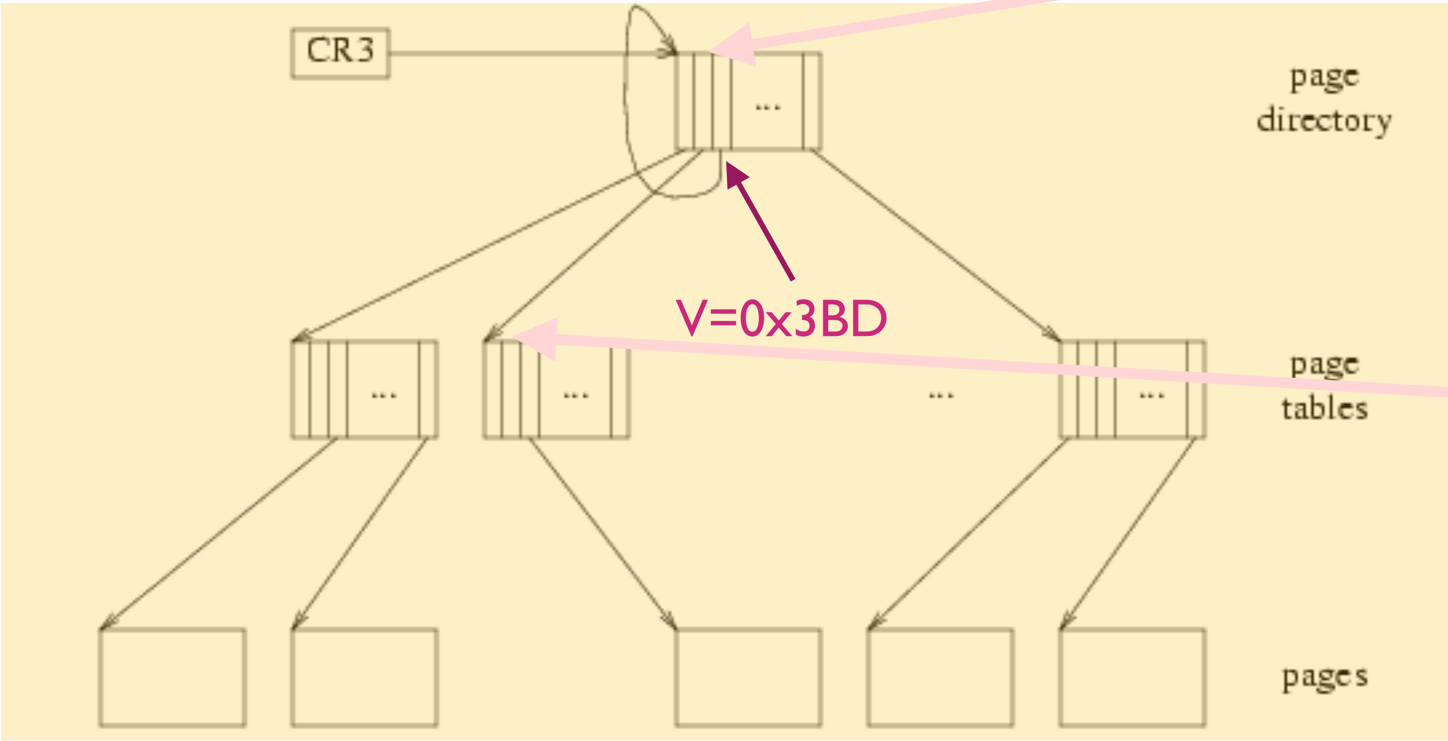
- Pgdir and page tables are referenced by physical memory
- Would be nice to have a single-level page table in order to read/update the page table
- Would be nice to have an easy way to find the virtual address of the page table
- Idea: Set up a special pgdir entry that allows us to:
  - Easily map to the physical pgdir
  - Easily get a contiguous map of the PTEs

# JOS UVPT (Virtual Page Table)



```
pd = lcr3();
pt = *(pd+4*PDX);
page = *(pt+4*PTX);
```

What MMU does



```
uvpd = 0x3BD<<22 + 0x3BD<<12
pd = lcr3();
pt = *(pd+4*0x3BD);
page = *(pt+4*0x3BD);
```

Finding the pgdir

```
pde_t pde = uvpd[513]
```

PTDE for page dir with PDX 513

```
uvpt = 0x3BD<<22 + 0x001<<12
pd = lcr3();
pt = *(pd+4*0x3BD);
page = *(pt+4*0x001);
```

Finding the page table entries

```
pte_t pte = uvpt[6532]
```

PTE for frame number 6532