6.828: Using Virtual Memory

Adam Belay

abelay@mit.edu

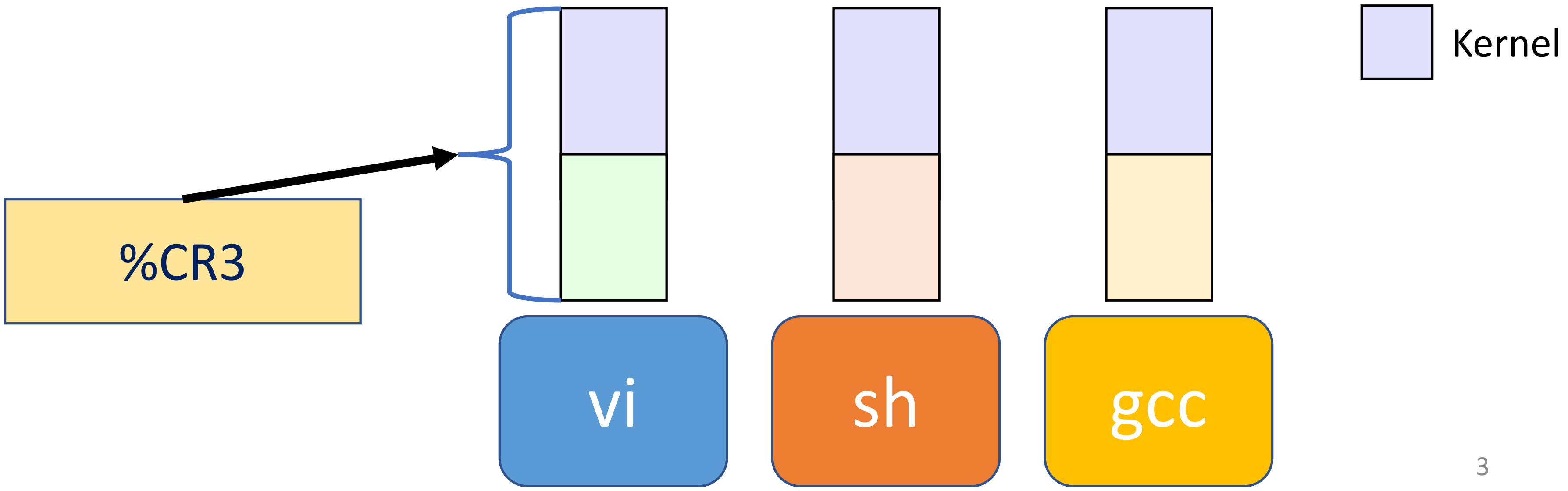
Outline

Cool things you can do with virtual memory:

- Lazy page allocation (homework)
- Better performance/efficiency
  - E.g. One zero-filled page
  - E.g. Copy-on-write w/ fork()
- New features
  - E.g. Memory-mapped files
- This lecture may generate final project ideas

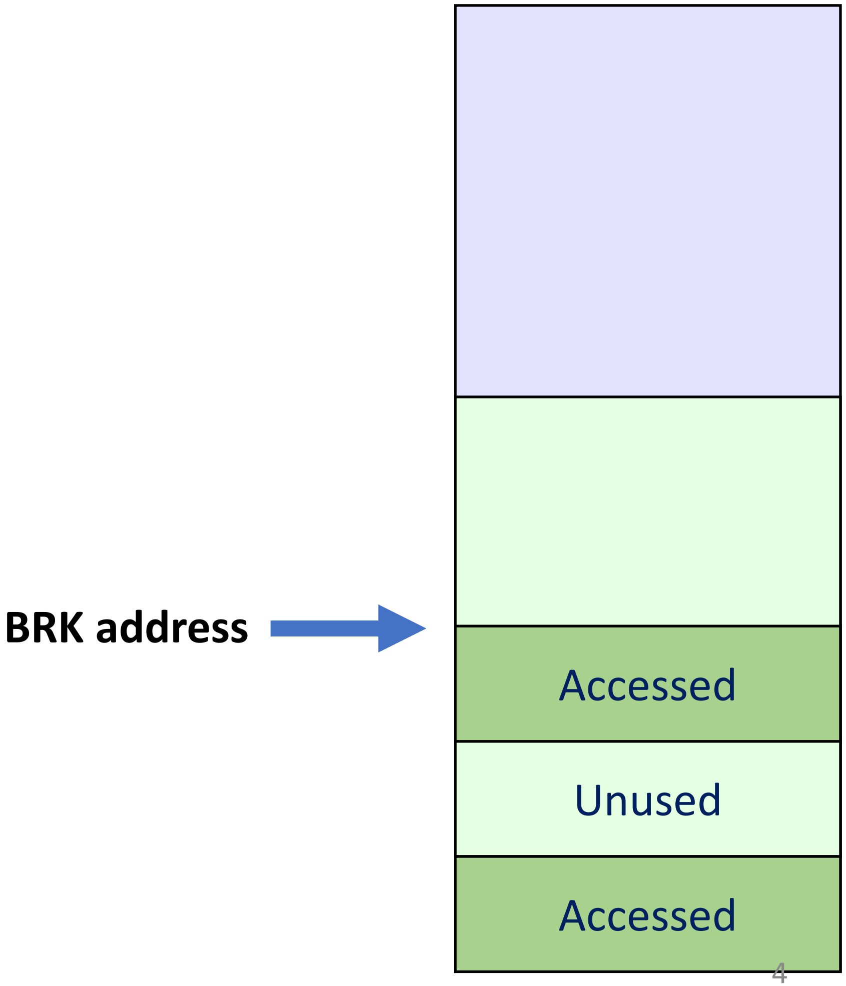
Recap: Virtual memory

- Primary goal: Isolation – each process has its own address space
- But… virtual memory provides a level of indirection that allows the kernel to do cool stuff



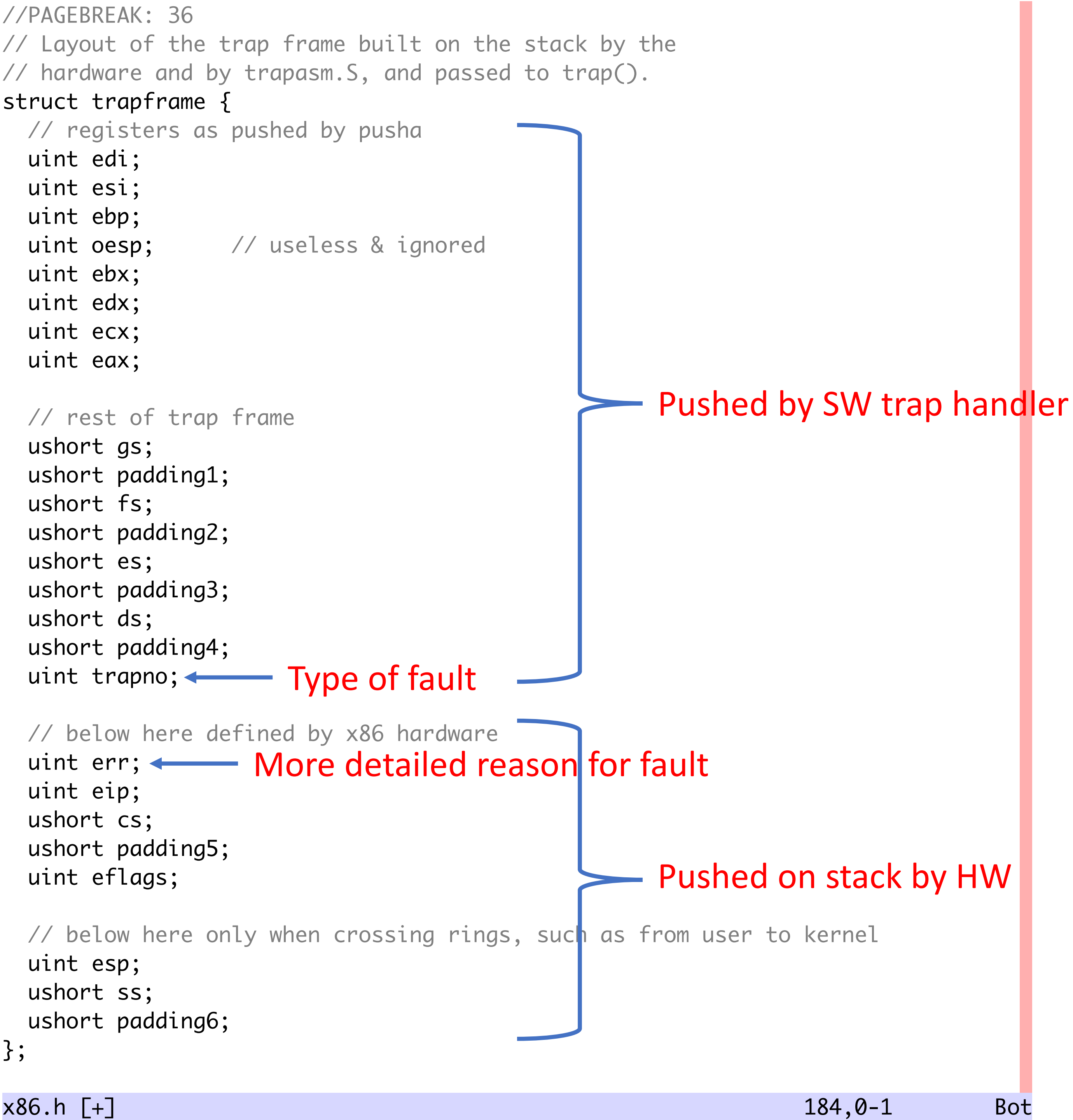
Homework: On-demand page allocation

- Problem: sbrk() is old-fashioned
  - Allocates memory that may never be used
- Modern OSes allocate memory lazily
  - Insert physical pages when they're accessed instead of in advance



x86 page faults

- x86 supports few dozen or so exceptions, one of them is T_PGFLT
- Exceptions are controlled transfers into the kernel
- Information we might need to handle a page fault:
  1. The VA that caused the fault
  2. The type of violation that caused the fault
  3. The EIP and CPL when the fault occurred



```
//PAGEBREAK: 36
// 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;
};
x86.h [+]                                                184,0-1        Bot
```

Pushed by SW trap handler

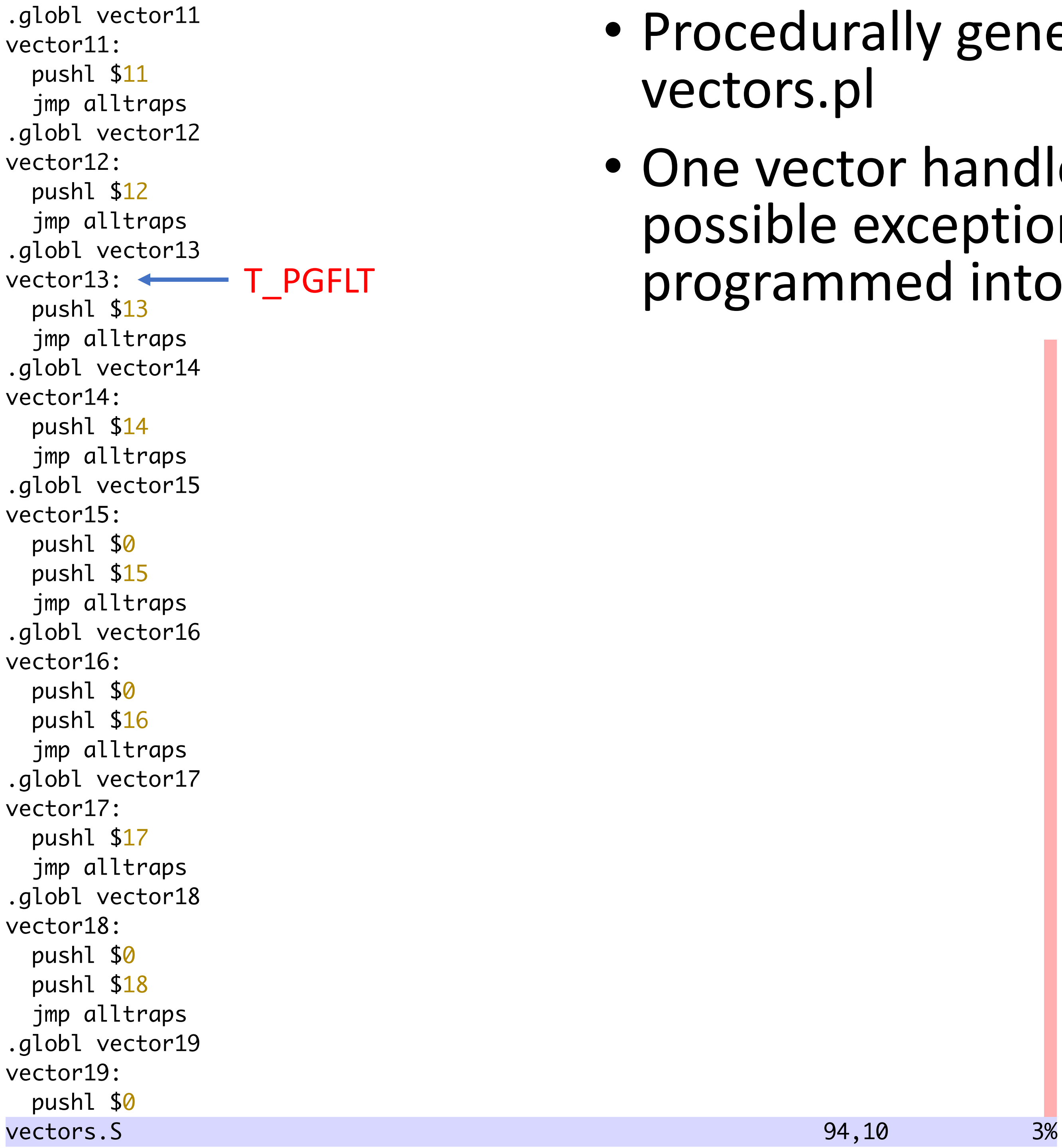
Type of fault

More detailed reason for fault

Pushed on stack by HW

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, HW pushes EIP, CS, EFLAGs, etc.



```
.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
vector15:
  pushl $0
  pushl $15
  jmp alltraps
.globl vector16
vector16:
  pushl $0
  pushl $16
  jmp alltraps
.globl vector17
vector17:
  pushl $17
  jmp alltraps
.globl vector18
vector18:
  pushl $0
  pushl $18
  jmp alltraps
.globl vector19
vector19:
  pushl $0
vectors.S                                                          94,10          3%
```

T_PGFLT

- Procedurally generated by vectors.pl
- One vector handler for each possible exception, each programmed into IDT

```
#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 and per-cpu segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es
  movw $(SEG_KCPU<<3), %ax
  movw %ax, %fs
  movw %ax, %gs

  # 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
trapasm.S                                                    1,1           All
```

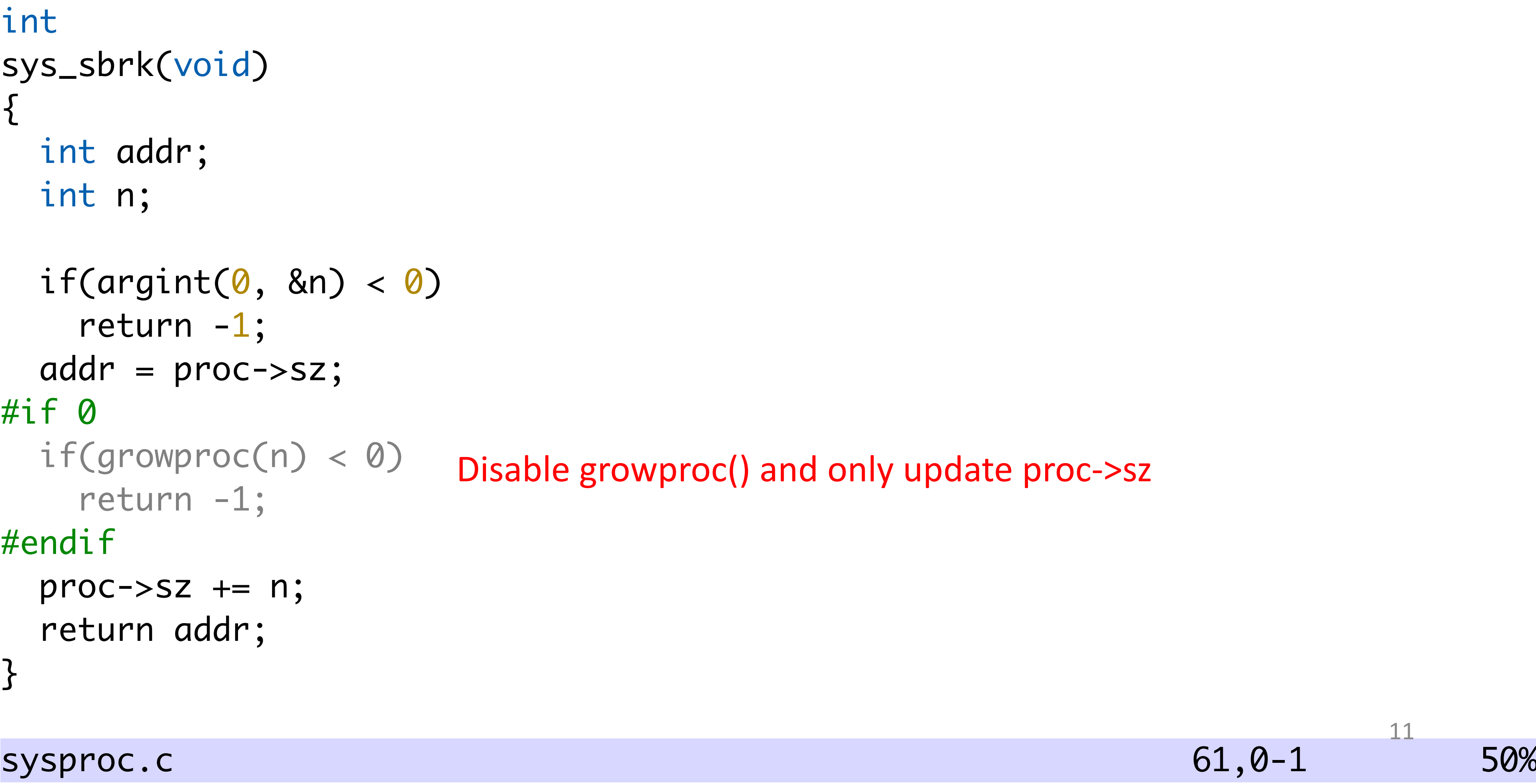
Construct SW portion of trap frame

Enter kernel C code

Gathering information to handle a page fault

1. The VA that caused the fault
    - movl %cr2, %ecx, or rcr2() in xv6
2. 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 a write
    - **FEC_U**: page fault occurred while in user mode
3. The EIP and CPL where the fault occurred
    - **EIP**: tf->eip
    - **CPL**: (tf->cs & 0x3) > 0 or check for (tf->err & FEC_U) > 0

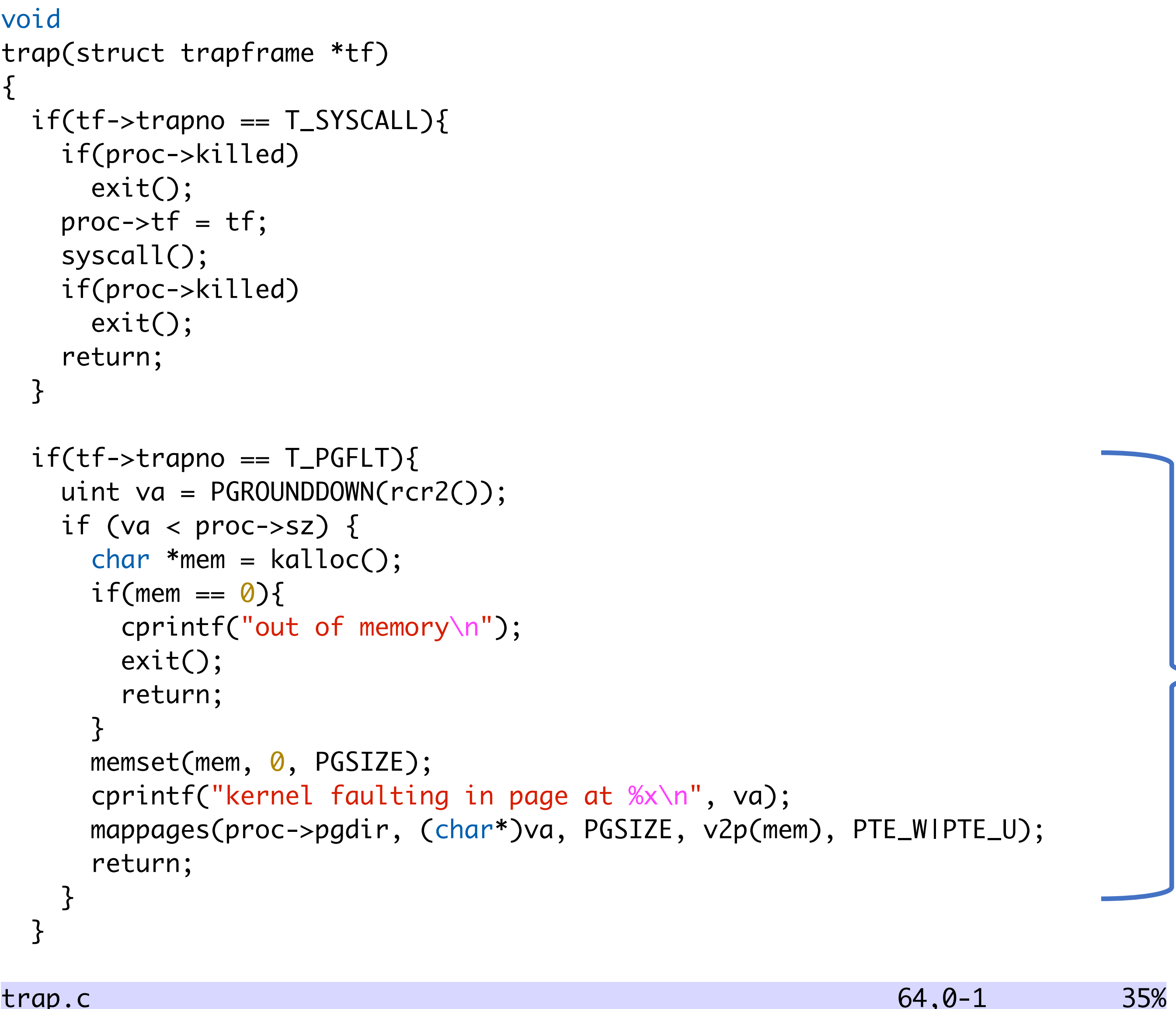
HW Solution: Changes to sys_sbrk()



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

  if(argint(0, &n) < 0)
    return -1;
  addr = proc->sz;
#if 0
  if(growproc(n) < 0)
    return -1;
#endif
  proc->sz += n;
  return addr;
}
sysproc.c                                                    61,0-1          50%
```

Disable growproc() and only update proc->sz

HW Solution: Changes to trap()



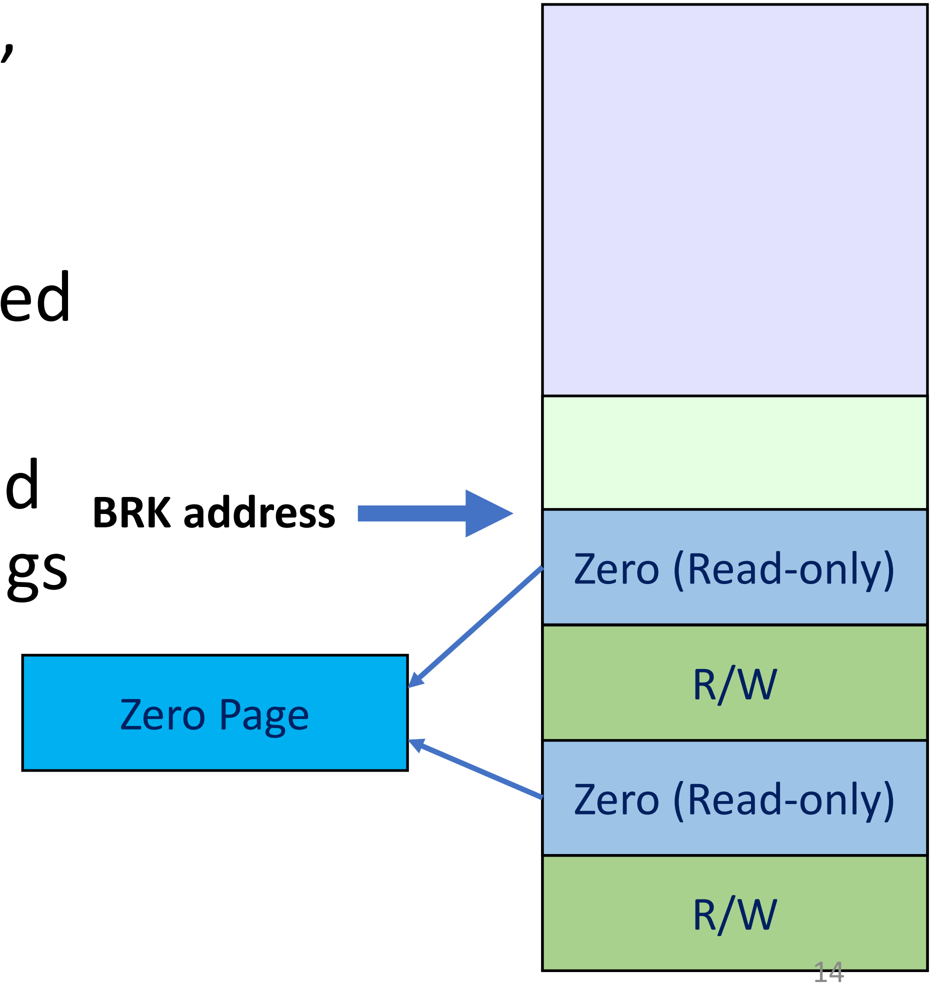
```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(proc->killed)
      exit();
    proc->tf = tf;
    syscall();
    if(proc->killed)
      exit();
    return;
  }

  if(tf->trapno == T_PGFLT){
    uint va = PGROUNDDOWN(rcr2());
    if (va < proc->sz) {
      char *mem = kalloc();
      if(mem == 0){
        cprintf("out of memory\n");
        exit();
        return;
      }
      memset(mem, 0, PGSIZE);
      cprintf("kernel faulting in page at %x\n", va);
      mappages(proc->pgdir, (char*)va, PGSIZE, v2p(mem), PTE_W|PTE_U);
      return;
    }
  }
trap.c                                                    64,0-1          35%
```

New T_PGFLT handler

On-demand page allocation demo

Optimization: Zero pages

- Observation: In practice, some memory is never written to
- All memory gets initialized to zero
- Idea: Use just **one** zeroed page for all zero mappings
- Copy the zero page on write



BRK address

Zero (Read-only)

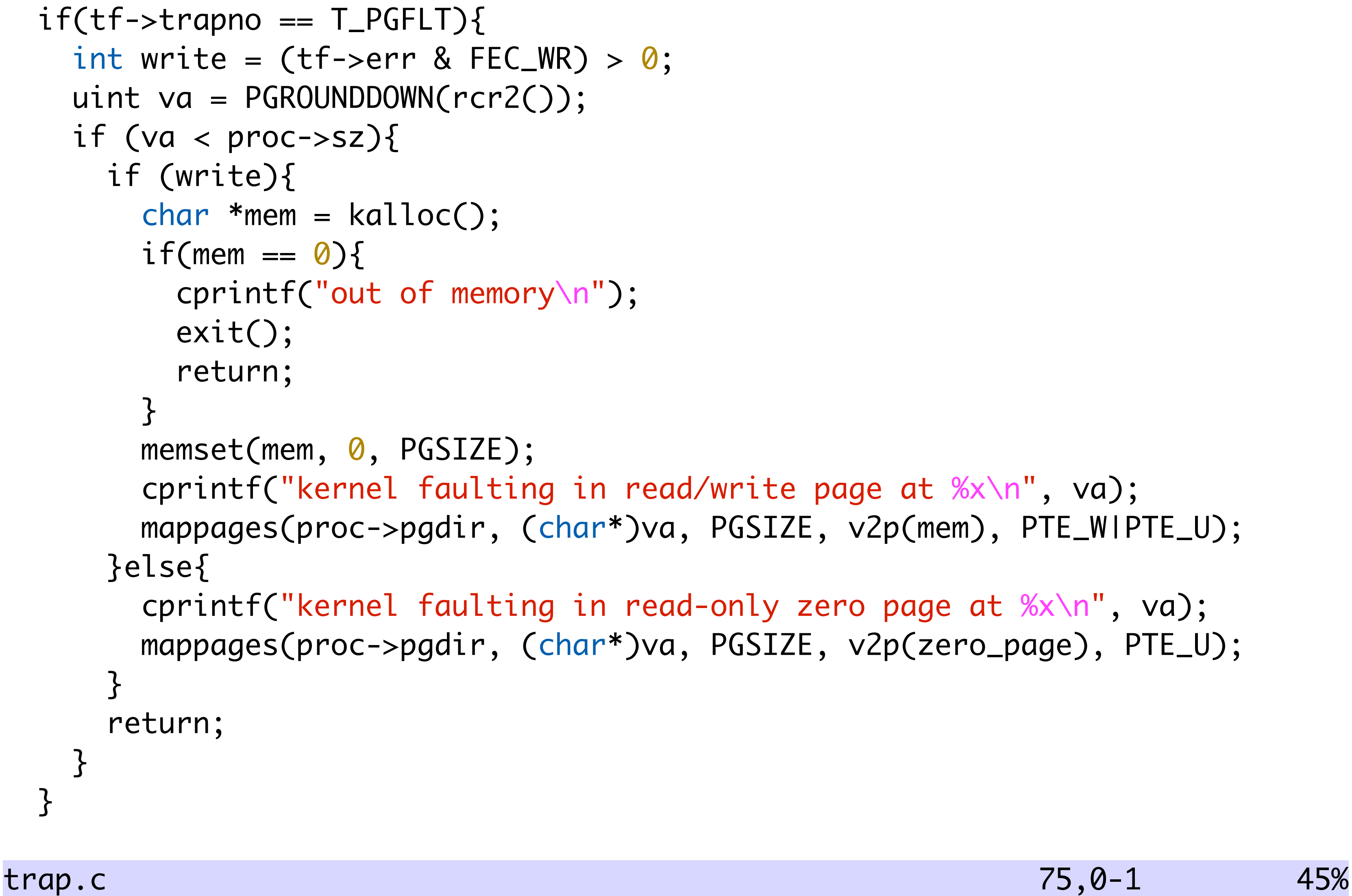
R/W

Zero (Read-only)

R/W

Zero Page

Zero page support: Changes to trap()



```
if(tf->trapno == T_PGFLT){
  int write = (tf->err & FEC_WR) > 0;
  uint va = PGROUNDDOWN(rcr2());
  if (va < proc->sz){
    if (write){
      char *mem = kalloc();
      if(mem == 0){
        cprintf("out of memory\n");
        exit();
        return;
      }
      memset(mem, 0, PGSIZE);
      cprintf("kernel faulting in read/write page at %x\n", va);
      mappages(proc->pgdir, (char*)va, PGSIZE, v2p(mem), PTE_W|PTE_U);
    }else{
      cprintf("kernel faulting in read-only zero page at %x\n", va);
      mappages(proc->pgdir, (char*)va, PGSIZE, v2p(zero_page), PTE_U);
    }
    return;
  }
}
trap.c                                                      75,0-1        45%
```

Zeroed page allocation demo

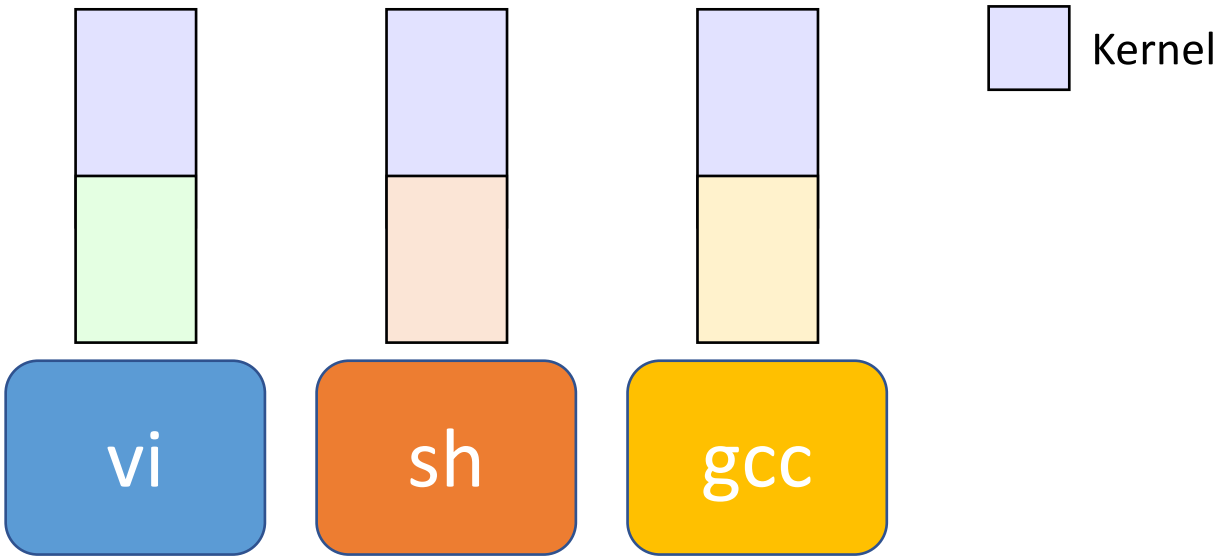
Caveats

- Page faults below user stack are invalid
- Negative ’n’ argument to sbrk() doesn’t remove mappings
- What about fork()?

- Real kernels are difficult to build, every detail matters

Optimization: Share kernel page mappings

- Observation: Every page table has identical kernel mappings
- Idea: Share kernel level 2 tables across all page tables



Feature: Stack guard pages

- Observation: Stack has a finite size
- Push too much data and it could overflow into adjacent memory
- Idea: Install an empty mapping (PTE_P cleared) at the bottom of the stack
- Could automatically increase stack size in page fault handler

Optimization: Copy-on-write fork()

- Observation: Fork() copies all pages in new process
- But often, exec() is called immediately after fork()
  - Wasted copies
- 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 object file into memory
  - Expensive, requires slow disk block access
  - Maybe not all of the file will be used
- Idea: Mark mapping as demand paged
  - On page fault, read disk block and install PTE
- Challenge: What if file is larger than physical memory?

Feature: Support more virtual memory than physical RAM

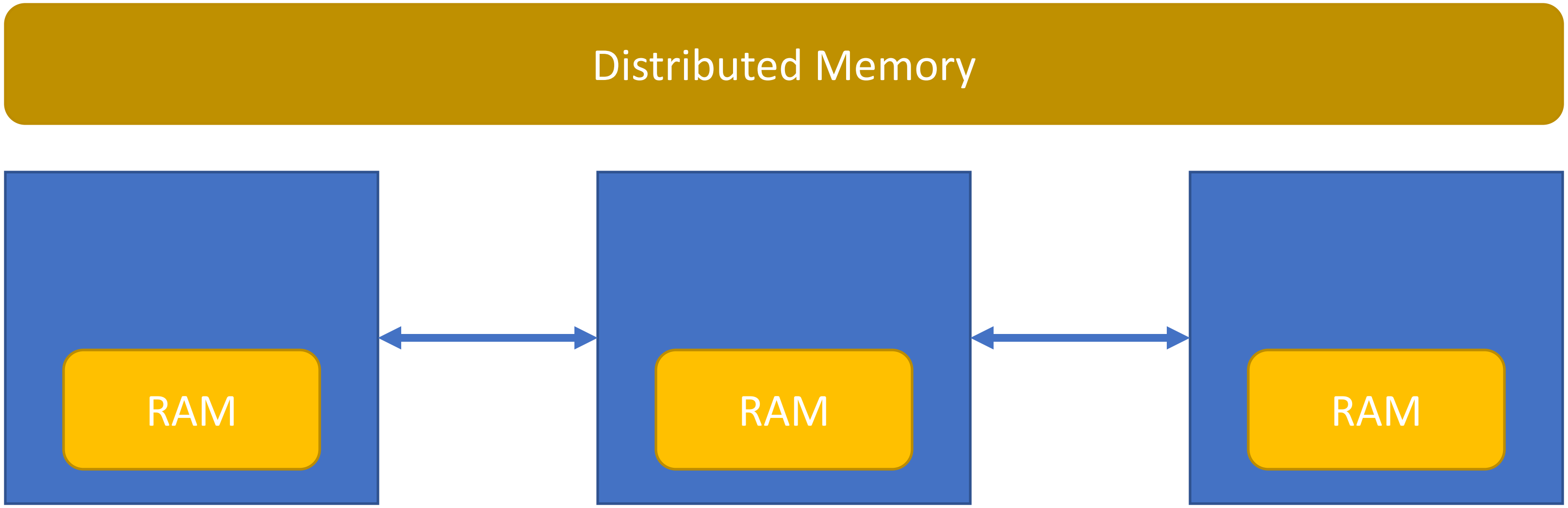
- Observation: More disk capacity than RAM
- Idea: “Page in” and out data between disk and RAM
    - Use page table entries to detect when disk access is needed
    - Use page table to find least recently used disk blocks to write back
- Works well when working set fits in RAM

Feature: Memory-mapped files

- Normally files accessed through read(), write(), and lseek()
- Idea: Use load and store to access file instead
  - New system call mmap() can place file at location in memory
  - Use memory offset to select block rather than seeking

Feature: Distributed shared memory

- Idea: Use virtual memory to pretend that physical memory is shared between several machines on the network



JOS virtual memory layout

Conclusion

- There’s no one way to design an OS
  - Many OSes use virtual memory
  - But you don’t have to!
- xv6 and JOS present two examples of OS design
  - They lack many features of real OSes
  - But still quite complex!
- Our goal: Teach you ideas so you can extrapolate