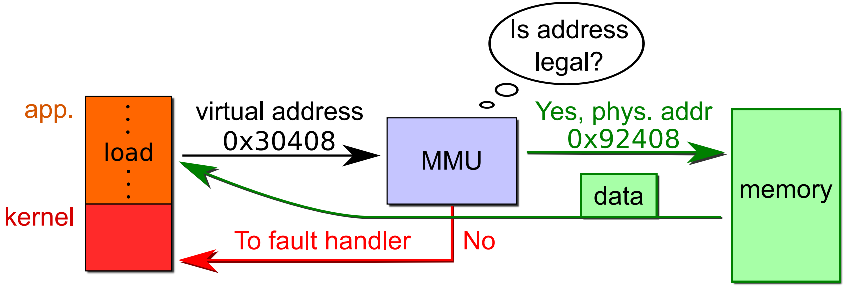
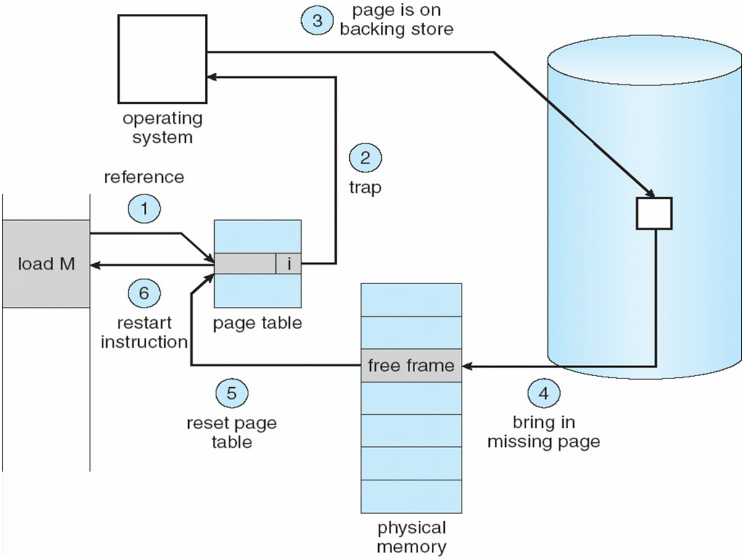
Virtual memory goals



- Give each program its own “virtual” address space
  - At run time, Memory-Management Unit relocates each load, store to actual memory… App doesn’t see physical memory
- Enforces protection
- Allows programs to see more memory than exists

Paging

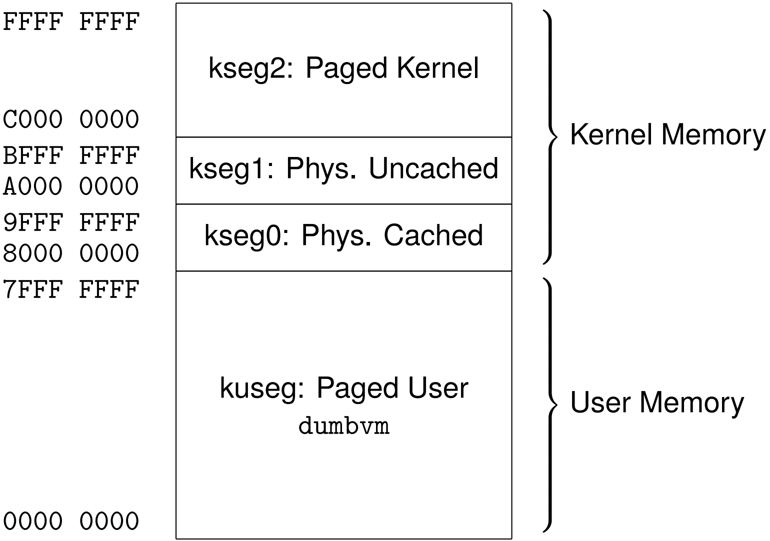


- Use disk to simulate larger virtual than physical mem

Outline

1. OS/161 Virtual Memory
2. User-level API
3. Virtual Memory Implementation
4. Case study: 4.4 BSD

MIPS Memory Layout



OS/161 dumbvm Address Translation

```
struct addrspace {
  vaddr_t as_vbase1; /* Segment 1 */
  paddr_t as_pbase1;
  size_t as_npages1;
  vaddr_t as_vbase2; /* Segment 2 */
  paddr_t as_pbase2;
  size_t as_npages2;
  paddr_t as_stackpbase; /* Stack Base */
};
```

- Implements segments with a TLB!
- Three segments code, data, and a fixed stack
- Virtual base (vbase), Physical base (pbase), and Number of pages (npages)
- Stack has a physical base address

OS/161 dumbvm Segments

- *vbase*: Virtual Base Address
- $vtop = vbase + npages * PAGE\_SIZE$: Virtual Top Address
- Segment maps memory between vbase and vtop

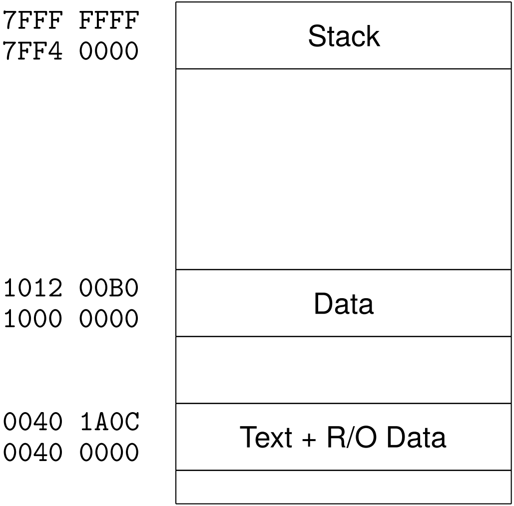
- *pbase*: Physical Base Address
- $paddr = faddr - vbase + pbase$: Convert Physical to Virtual

- Stack is always 12 pages in size
- Grows down from the top of memory

- Looks a like like the original UNIX releases in the 80s
- Assignment 3 you will replace this with a RADIX tree (similar to x86)

OS/161 Memory Layout: user/testbin/sort

- Example: `vbase1=0x400000`, `npages1=0x2`, `pbase1=XXXXXXXX`, `vbase2=0x10000000`, `npages2=0x12`, `pbase2=YYYYYYYY`, `stackpbase=ZZZZZZZZ`



OS/161 dumbvm Translation Logic

```
// USERSTACK=0x8000_0000, DUMBVM_STACKPAGES=12
// PAGE_SIZE=4K
vbase1 = as->as_vbase1;
vtop1 = vbase1 + as->as_npages1 * PAGE_SIZE;
vbase2 = as->as_vbase2;
vtop2 = vbase2 + as->as_npages2 * PAGE_SIZE;
stackbase = USERSTACK - DUMBVM_STACKPAGES * PAGE_SIZE;
stacktop = USERSTACK;

if (faultaddr >= vbase1 && faultaddr < vtop1) {
  paddr = (faultaddr - vbase1) + as->as_pbase1;
} else if (faultaddr >= vbase2 && faultaddr < vtop2) {
  paddr = (faultaddr - vbase2) + as->as_pbase2;
} else if (faultaddr >= stackbase && faultaddr < stacktop) {
  paddr = (faultaddr - stackbase) + as->as_stackpbase;
} else {
  return EFAULT;
}
```

OS/161 Details

- TLB fault exception calls:
  - `common_exception` pushes trap frame
  - `mips_trap()` determines trap cause and calls `vm_fault()`
  - `vm_fault()` computes physical address from faulting address
  - Calls `tlb_write()` to update the TLB and returns
- Address Spaces APIs
  - `as_define_region()` creates a segment (2 max)
  - `as_activate()` invalidates the TLB
  - `as_copy()` duplicates the entire process

OS/161 and ELF: readelf

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 09 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            FreeBSD
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices x86-64
  Version:                           0x1
  Entry point address:               0x203000
  Start of program headers:          64 (bytes into file)
  Start of section headers:          38416 (bytes into file)
  Flags:                             0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         10
  Size of section headers:           64 (bytes)
  Number of section headers:         29
  Section header string table index: 28
```

- Describes the binary and starting point (entry point)

OS/161 and ELF: readelf

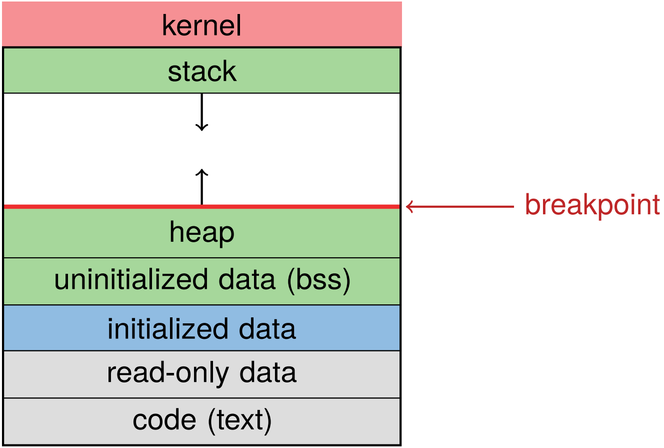
```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flg    Align
  PHDR           0x0000000000000040 0x0000000000200040 0x0000000000200040
                 0x0000000000000230 0x0000000000000230 R    0x8
  INTERP         0x0000000000000270 0x0000000000200270 0x0000000000200270
                 0x0000000000000015 0x0000000000000015 R    0x1
      [Requesting program interpreter: /libexec/ld-elf.so.1]
  LOAD           0x0000000000000000 0x0000000000200000 0x0000000000200000
                 0x0000000000002bd8 0x0000000000002bd8 R    0x1000
  LOAD           0x0000000000003000 0x0000000000203000 0x0000000000203000
                 0x0000000000004b50 0x0000000000004b50 R E  0x1000
  LOAD           0x0000000000008000 0x0000000000208000 0x0000000000208000
                 0x00000000000011a0 0x000000000000229c RW   0x1000
  ...
```

- Program Headers are instructions for the OS
- `INTERP` is the dynamic linker (more in a later class)
- `LOAD` is a single segment for the OS to load
- Shown is /bin/ls from FreeBSD and it has more than two segments

Outline

1. OS/161 Virtual Memory
2. User-level API
3. Virtual Memory Implementation
4. Case study: 4.4 BSD

Recall typical virtual address space

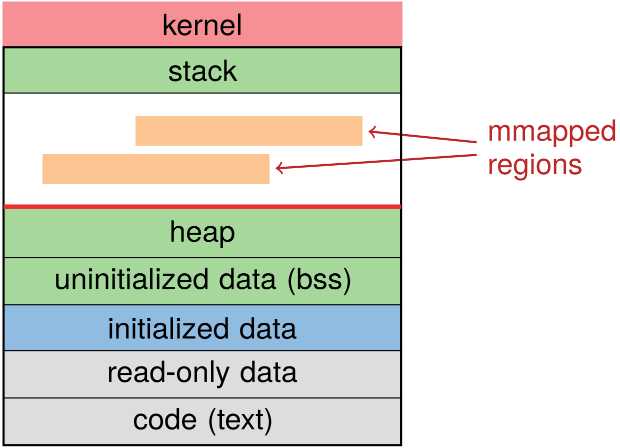


- Dynamically allocated memory goes in heap
- Top of heap called *breakpoint*
  - Addresses between breakpoint and stack all invalid

Early VM system calls

- OS keeps "Breakpoint" – top of heap
  - Memory regions between breakpoint & stack fault on access
- `char *brk (const char *addr);`
  - Set and return new value of breakpoint
- `char *sbrk (int incr);`
  - Increment value of the breakpoint & return old value
- Can implement `malloc` in terms of `sbrk`
  - But hard to "give back" physical memory to system

Memory mapped files



- Other memory objects between heap and stack

`mmap` system call

- `void *mmap (void *addr, size_t len, int prot,`
  `int flags, int fd, off_t offset)`
  - Map file specified by `fd` at virtual address `addr`
  - If `addr` is NULL, let kernel choose the address
- `prot` – protection of region
  - OR of PROT_EXEC, PROT_READ, PROT_WRITE, PROT_NONE
- `flags`
  - MAP_ANON – anonymous memory (`fd` should be -1)
  - MAP_PRIVATE – modifications are private
  - MAP_SHARED – modifications seen by everyone

More VM system calls

- `int msync(void *addr, size_t len, int flags);`
  - Flush changes of mmapped file to backing store
- `int munmap(void *addr, size_t len)`
  - Removes memory-mapped object
- `int mprotect(void *addr, size_t len, int prot)`
  - Changes protection on pages to or of `PROT_...`
- `int mincore(void *addr, size_t len, char *vec)`
  - Returns in `vec` which pages present

Exposing page faults

```
struct sigaction {
  union {                 /* signal handler */
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
  };
  sigset_t sa_mask;       /* signal mask to apply */
  int sa_flags;
};

int sigaction (int sig, const struct sigaction *act,
               struct sigaction *oact)
```

- Can specify function to run on `SIGSEGV`
  (Unix signal raised on invalid memory access)

Example: OpenBSD/i386 siginfo

```
struct sigcontext {
  int sc_gs; int sc_fs; int sc_es; int sc_ds;
  int sc_edi; int sc_esi; int sc_ebp; int sc_ebx;
  int sc_edx; int sc_ecx; int sc_eax;

  int sc_eip; int sc_cs; /* instruction pointer */
  int sc_eflags;         /* condition codes, etc. */
  int sc_esp; int sc_ss; /* stack pointer */

  int sc_onstack;        /* sigstack state to restore */
  int sc_mask;           /* signal mask to restore */

  int sc_trapno;
  int sc_err;
};
```

- Linux uses `ucontext_t` – same idea, just uses nested structures that won't all fit on one slide

VM tricks at user level

- Combination of `mprotect`/`sigaction` very powerful
  - Can use OS VM tricks in user-level programs [Appel]
  - E.g., fault, unprotect page, return from signal handler
- Technique used in object-oriented databases
  - Bring in objects on demand
  - Keep track of which objects may be dirty
  - Manage memory as a cache for much larger object DB
- Other interesting applications
  - Useful for some garbage collection algorithms
  - Snapshot processes (copy on write)

Outline

1. OS/161 Virtual Memory
2. User-level API
3. Virtual Memory Implementation
4. Case study: 4.4 BSD

Overview

- Windows and Most UNIX systems seperate the VM system into two parts
  - *VM PMap*: Manages the hardware interface (e.g. TLB in MIPS)
  - *VM Map*: Machine independent representation of memory
- VM Map consists of one or more *objects* (or *segments*)
- Each object consists of a contiguous `mmap()`
- Objects can be backed by files and/or shared between processes
- VM PMap manages the hardware (often caches mappings)

Operation

- Calls into `mmap()`, `munmap()`, `mprotect()`
  - Update VM Map
  - VM Map routines call into the VM PMap to invalidate and update the TLB
- Page faults
  - Exception handler calls into the VM PMap to load the TLB
  - If the page isn't in the PMap we call VM Map code
- Low memory options
  - PMap is a cache and can be discarded during a low memory condition

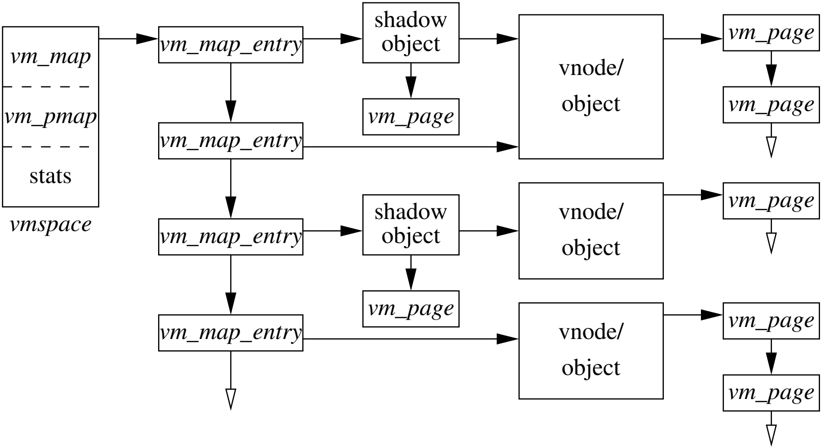
Outline

1. OS/161 Virtual Memory
2. User-level API
3. Virtual Memory Implementation
4. Case study: 4.4 BSD

4.4 BSD VM system [McKusick]

- Each process has a *vmspace* structure containing
  - *vm_map* – machine-independent virtual address space
  - *vm_pmap* – machine-dependent data structures
  - statistics – e.g. for syscalls like *getrusage ()*
- *vm_map* is a linked list of *vm_map_entry* structs
  - *vm_map_entry* covers contiguous virtual memory
  - points to *vm_object* struct
- *vm_object* is source of data
  - e.g. vnode object for memory mapped file
  - points to list of *vm_page* structs (one per mapped page)
  - *shadow objects* point to other objects for copy on write

4.4 BSD VM data structures



Pmap (machine-dependent) layer

- Pmap layer holds architecture-specific VM code
- VM layer invokes pmap layer
  - On page faults to install mappings
  - To protect or unmap pages
  - To ask for dirty/accessed bits
- Pmap layer is lazy and can discard mappings
  - No need to notify VM layer
  - Process will fault and VM layer must reinstall mapping
- Pmap handles restrictions imposed by cache

Example uses

- *vm_map_entry* structs for a process
  - r/o text segment $\rightarrow$ file object
  - r/w data segment $\rightarrow$ shadow object $\rightarrow$ file object
  - r/w stack $\rightarrow$ anonymous object
- New *vm_map_entry* objects after a fork:
  - Share text segment directly (read-only)
  - Share data through two new shadow objects (must share pre-fork but not post-fork changes)
  - Share stack through two new shadow objects
- Must discard/collapse superfluous shadows
  - E.g., when child process exits

What happens on a fault?

- Traverse *vm_map_entry* list to get appropriate entry
  - No entry? Protection violation? Send process a SIGSEGV
- Traverse list of [shadow] objects
- For each object, traverse *vm_page* structs
- Found a *vm_page* for this object?
  - If first *vm_object* in chain, map page
  - If read fault, install page read only
  - Else if write fault, install copy of page
- Else get page from object
  - Page in from file, zero-fill new page, etc.

Paging in day-to-day use

- Demand paging
  - Read pages from *vm_object* of executable file
- Copy-on-write (`fork`, `mmap`, etc.)
  - Use shadow objects
- Growing the stack, BSS page allocation
  - A bit like copy-on-write for `/dev/zero`
  - Can have a single read-only zero page for reading
  - Special-case write handling with pre-zeroed pages
- Shared text, shared libraries
  - Share *vm_object* (shadow will be empty where read-only)
- Shared memory
  - Two processes `mmap` same file, have same *vm_object* (no shadow)