Address translation and sharing using segments

This lecture is about virtual memory, focusing on address spaces. It is the first lecture out of series of lectures that uses xv6 as a case study.

Address spaces

- OS: kernel program and user-level programs. For fault isolation each program runs in a separate address space. The kernel address spaces is like user address spaces, expect it runs in kernel mode. The program in kernel mode can execute priviledge instructions (e.g., writing the kernel's code segment registers).
- One job of kernel is to manage address spaces (creating, growing, deleting, and switching between them)
    - Each address space (including kernel) consists of the binary representation for the text of the program, the data part part of the program, and the stack area.
    - The kernel address space runs the kernel program. In a monolithic organization the kernel manages all hardware and provides an API to user programs.
    - Each user address space contains a program. A user progam may ask to shrink or grow its address space.
- The main operations:
    - Creation. Allocate physical memory to storage program. Load program into physical memory. Fill address spaces with references to physical memory.
    - Growing. Allocate physical memory and add it to address space.
    - Shrinking. Free some of the memory in an address space.
    - Deletion. Free all memory in an address space.
    - Switching. Switch the processor to use another address space.
    - Sharing. Share a part of an address space with another program.

Two main approaches to implementing address spaces: using segments and using page tables. Often when one uses segments, one also uses page tables. But not the other way around; i.e., paging without segmentation is common.

Example support for address spaces: x86

For an operating system to provide address spaces and address translation typically requires support from hardware. The translation and checking of permissions typically must happen on each address used by a program, and it would be too slow to check that in software (if even possible). The division of labor is operating system manages address spaces, and hardware translates addresses and checks permissions.

PC block diagram without virtual memory support:

- physical address
- base, IO hole, extended memory
- Physical address == what is on CPU's address pins

The x86 starts out in real mode and translation is as follows:

- segment*16+offset ==> physical address
- no protection: program can load anything into seg reg

The operating system can switch the x86 to protected mode, which allows the operating system to create address spaces. Translation in protected mode is as follows:

- selector:offset (logical addr)
  ==SEGMENTATION==>
- linear address
  ==PAGING ==>
- physical address

Next lecture covers paging; now we focus on segmentation.

Protected-mode segmentation works as follows:

- protected-mode segments add 32-bit addresses and protection
  - wait: what's the point? the point of segments in real mode was bigger addresses, but 32-bit mode fixes that!
- segment register holds segment selector
- selector indexes into global descriptor table (GDT)
- segment descriptor holds 32-bit base, limit, type, protection
- la = va + base ; assert(va < limit);
- seg register usually implicit in instruction
  - DS:REG
    - `movl $0x1, _flag`
  - SS:ESP, SS:EBP
    - `pushl %ecx, pushl $_i`
    - `popl %ecx`
    - `movl 4(%ebp),%eax`
  - CS:EIP
    - instruction fetch
  - String instructions: read from DS:ESI, write to ES:EDI
    - `rep movsb`
  - Exception: far addresses
    - `ljmp $selector, $offset`
- LGDT instruction loads CPU's GDT register
- you turn on protected mode by setting PE bit in CR0 register
- what happens with the next instruction? CS now has different meaning...

- How to transfer from segment to another, perhaps with different priveleges.
  - Current privilege level (CPL) is in the low 2 bits of CS
  - CPL=0 is privileged O/S, CPL=3 is user
  - Within in the same privelege level: ljmp.
  - Transfer to a segment with more privilege: call gates.
    - a way for app to jump into a segment and acquire privs
    - CPL must be <= descriptor's DPL in order to read or write segment
    - call gates can change privelege **and** switch CS and SS segment
    - call gates are implemented using a special type segment descriptor in the GDT.
    - interrupts are conceptually the same as call gates, but their descriptor is stored in the IDT. We will use interrupts to transfer control between user and kernel mode, both in JOS and xv6. We will return to this in the lecture about interrupts and exceptions.
- What about protection?
  - can o/s limit what memory an application can read or write?
  - app can load any selector into a seg reg...
  - but can only mention indices into GDT
  - app can't change GDT register (requires privilege)
  - why can't app write the descriptors in the GDT?
  - what about system calls? how to they transfer to kernel?
  - app cannot **just** lower the CPL

Case study (xv6)

xv6 is a reimplementation of Unix 6th edition.

- v6 is a version of the orginal Unix operating system for DEC PDP11
  - PDP-11 (1972):
  - 16-bit processor, 18-bit physical (40)
  - UNIBUS
  - memory-mapped I/O
  - performance: less than 1MIPS
  - register-to-register transfer: 0.9 usec
  - 56k-228k (40)
  - no paging, but some segmentation support
  - interrupts, traps
  - about $10K
  - rk disk with 2MByte of storage
  - with cabinet 11/40 is 400lbs
- Unix v6
  - Unix papers.
  - 1976; first widely available Unix outside Bell labs
  - Thompson and Ritchie
  - Influenced by Multics but simpler.
  - complete (used for real work)

    - Multi-user, time-sharing
    - small (43 system calls)
    - modular (composition through pipes; one had to split programs!!)
    - compactly written (2 programmers, 9,000 lines of code)
    - advanced UI (shell)
    - introduced C (derived from B)
    - distributed with source
    - V7 was sold by Microsoft for a couple years under the name Xenix
- Lion's commentary
    - surpressed because of copyright issue
    - resurfaced in 1996
- xv6 written for 6.828:
    - v6 reimplementation for x86
    - does't include all features of v6 (e.g., xv6 has 20 of 43 system calls).
    - runs on symmetric multiprocessing PCs (SMPs).

Newer Unixs have inherited many of the conceptual ideas even though they added paging, networking, graphics, improve performance, etc.

You will need to read most of the source code multiple times. Your goal is to explain every line to yourself.

Overview of address spaces in xv6

In today's lecture we see how xv6 creates the kernel address spaces, first user address spaces, and switches to it. To understand how this happens, we need to understand in detail the state on the stack too---this may be surprising, but a thread of control and address space are tightly bundled in xv6, in a concept called *process*. The kernel address space is the only address space with multiple threads of control. We will study context switching and process management in detail next weeks; creation of the first user process (init) will get you a first flavor.

xv6 uses only the segmentation hardware on xv6, but in a limited way. (In JOS you will use page-table hardware too, which we cover in next lecture.) The adddress space layouts are as follows:

- In kernel address space is set up as follows:
- `    the code segment runs from 0 to 2^32 and is mapped X and R`
- `    the data segment runs from 0 to 2^32 but is mapped W (read and write).`

- For each process, the layout is as follows:
- `    text`
- `    original data and bss`
- `    fixed-size stack`
- `    expandable heap`

The text of a process is stored in its own segment and the rest in a data segment.

xv6 makes minimal use of the segmentation hardware available on the x86. What other plans could you envision?

In xv6, each each program has a user and a kernel stack; when the user program switches to the kernel, it switches to its kernel stack. Its kernel stack is stored in process's proc structure. (This is arranged through the descriptors in the IDT, which is covered later.)

xv6 assumes that there is a lot of physical memory. It assumes that segments can be stored contiguously in physical memory and has therefore no need for page tables.

xv6 kernel address space

Let's see how xv6 creates the kernel address space by tracing xv6 from when it boots, focussing on address space management:

- Where does xv6 start after the PC is power on: start (which is loaded at physical address 0x7c00; see lab 1).
- 1025-1033: are we in real mode?
    - how big are logical addresses?
    - how big are physical addresses?
    - how are addresses physical calculated?
    - what segment is being used in subsequent code?
    - what values are in that segment?
- 1068: what values are loaded in the GDT?
    - 1097: gdtr points to gdt
    - 1094: entry 0 unused
    - 1095: entry 1 (X + R, base = 0, limit = 0xffffffff, DPL = 0)
    - 1096: entry 2 (W, base = 0, limit = 0xffffffff, DPL = 0)
    - are we using segments in a sophisticated way? (i.e., controled sharing)
    - are P and S set?
    - are addresses translated as in protected mode when lgdt completes?
- 1071: no, and not even here.
- 1075: far jump, load 8 in CS. from now on we use segment-based translation.
- 1081-1086: set up other segment registers
- 1087: where is the stack which is used for procedure calls?
- 1087: cmain in the bootloader (see lab 1), which calls main0
- 1222: main0.
    - job of main0 is to set everthing up so that all xv6 convtions works
    - where is the stack? (sp = 0x7bec)
    - what is on it?

```
00007bec [00007bec]  7cda  // return address in cmain
00007bf0 [00007bf0]  0080  // callee-saved ebx
00007bf4 [00007bf4]  7369  // callee-saved esi
00007bf8 [00007bf8]  0000  // callee-saved ebp
```

    - `00007bfc [00007bfc]  7c49  // return address for cmain: spin`
    - `00007c00 [00007c00]  c031fcfa  // the instructions from 7c00 (start)`
- 1239-1240: switch to cpu stack (important for scheduler)
    - why -32?
    - what values are in ebp and esp?
    - `esp: 0x108d30   1084720`
    - `ebp: 0x108d5c   1084764`
    - what is on the stack?
    - `00108d30 [00108d30]  0000`
    - `00108d34 [00108d34]  0000`
    - `00108d38 [00108d38]  0000`
    - `00108d3c [00108d3c]  0000`
    - `00108d40 [00108d40]  0000`
    - `00108d44 [00108d44]  0000`
    - `00108d48 [00108d48]  0000`
    - `00108d4c [00108d4c]  0000`
    - `00108d50 [00108d50]  0000`
    - `00108d54 [00108d54]  0000`
    - `00108d58 [00108d58]  0000`
    - `00108d5c [00108d5c]  0000`
    - `00108d60 [00108d60]  0001`
    - `00108d64 [00108d64]  0001`
    - `00108d68 [00108d68]  0000`
    - `00108d6c [00108d6c]  0000`
    - what is 1 in 0x108d60? is it on the stack?
- 1242: is it save to reference bcpu? where is it allocated?
- 1260-1270: set up proc[0]
    - each process has its own stack (see struct proc).
    - where is its stack? (see the section below on physical memory management below).
    - what is the jmpbuf? (will discuss in detail later)
    - 1267: why -4?
- 1270: necessar to be able to take interrupts (will discuss in detail later)
- 1292: what process do you think scheduler() will run? we will study later how that happens, but let's assume it runs process0 on process0's stack.

xv6 user address spaces

- 1327: process0
    - process 0 sets up everything to make process conventions work out
    - which stack is process0 running? see 1260.
    - 1334: is the convention to release the proc_table_lock after being scheduled? (we will discuss locks later; assume there are no other processors for now.)
    - 1336: cwd is current working directory.
    - 1348: first step in initializing a template tram frame: set everything to zero. we are setting up process 0 as if it just entered the kernel from user

space and wants to go back to user space. (see x86.h to see what field have the value 0.)
    - 1349: why "|3"? instead of 0?
    - 1351: why set interrupt flag in template trapframe?
    - 1352: where will the user stack be in proc[0]'s address space?
    - 1353: makes a copy of proc0. fork() calls copyproc() to implement forking a process. This statement in essense is calling fork inside proc0, making a proc[1] a duplicate of proc[0]. proc[0], however, has not much in its address space of one page (see 1341).
        - 2221: grab a lock on the proc table so that we are the only one updating it.
        - 2116: allocate next pid.
        - 2228: we got our entry; release the lock. from now we are only modifying our entry.
        - 2120-2127: copy proc[0]'s memory. proc[1]'s memory will be identical to proc[0]'s.
        - 2130-2136: allocate a kernel stack. this stack is different from the stack that proc[1] uses when running in user mode.
        - 2139-2140: copy the template trapframe that xv6 had set up in proc[0].
        - 2147: where will proc[1] start running when the scheduler selects it?
        - 2151-2155: Unix semantics: child inherits open file descriptors from parent.
        - 2158: same for cwd.
    - 1356: load a program in proc[1]'s address space. the program loaded is the binary version of init.c (sheet 16).
    - 1374: where will proc[1] start?
    - 1377-1388: copy the binary into proc[1]'s address space. (you will learn about the ELF format in the labs.)
        - can the binary for init be any size for proc[1] to work correctly?
        - what is the layout of proc[1]'s address space? is it consistent with the layout described on line 1950-1954?
    - 1357: make proc[1] runnable so that the scheduler will select it to run. everything is set up now for proc[1] to run, "return" to user space, and execute init.
    - 1359: proc[0] gives up the processor, which calls sleep, which calls sched, which setjmps back to scheduler. let's peak a bit in scheduler to see what happens next. (we will return to the scheduler in more detail later.)
- 2219: this test will fail for proc[1]
- 2226: setupsegs(p) sets up the segments for proc[1]. this call is more interesting than the previous, so let's see what happens:
    - 2032-37: this is for traps and interrupts, which we will cover later.
    - 2039-49: set up new gdt.
    - 2040: why 0x100000 + 64*1024?
    - 2045: why 3? why is base p->mem? is p->mem physical or logical?

    - 2045-2046: how much the program for proc[1] be compiled if proc[1] will run successfully in user space?
    - 2052: we are still running in the kernel, but we are loading gdt. is this ok?
    - why have so few user-level segments? why not separate out code, data, stack, bss, etc.?
- 2227: record that proc[1] is running on the cpu
- 2228: record it is running instead of just runnable
- 2229: setjmp to fork_ret.
- 2282: which stack is proc[1] running on?
- 2284: when scheduled, first release the proc_table_lock.
- 2287: back into assembly.
- 2782: where is the stack pointer pointing to?
- `0020dfbc [0020dfbc]  0000`
- `0020dfc0 [0020dfc0]  0000`
- `0020dfc4 [0020dfc4]  0000`
- `0020dfc8 [0020dfc8]  0000`
- `0020dfcc [0020dfcc]  0000`
- `0020dfd0 [0020dfd0]  0000`
- `0020dfd4 [0020dfd4]  0000`
- `0020dfd8 [0020dfd8]  0000`
- `0020dfdc [0020dfdc]  0023`
- `0020dfe0 [0020dfe0]  0023`
- `0020dfe4 [0020dfe4]  0000`
- `0020dfe8 [0020dfe8]  0000`
- `0020dfec [0020dfec]  0000`
- `0020dff0 [0020dff0]  001b`
- `0020dff4 [0020dff4]  0200`
- `0020dff8 [0020dff8]  1000`
- 2783: why jmp instead of call?
- what will iret put in eip?
- what is 0x1b? what will iret put in cs?
- after iret, what will the processor being executing?

Managing physical memory

To create an address space we must allocate physical memory, which will be freed when an address space is deleted (e.g., when a user program terminates). xv6 implements a first-fit memory allocater (see kalloc.c).

It maintains a list of ranges of free memory. The allocator finds the first range that is larger than the amount of requested memory. It splits that range in two: one range of the size requested and one of the remainder. It returns the first range. When memory is freed, kfree will merge ranges that are adjacent in memory.

Under what scenarios is a first-fit memory allocator undesirable?

Growing an address space

How can a user process grow its address space? growproc.

- 2064: allocate a new segment of old size plus n
- 2067: copy the old segment into the new (ouch!)
- 2068: and zero the rest.
- 2071: free the old physical memory

We could do a lot better if segments didn't have to contiguous in physical memory. How could we arrange that? Using page tables, which is our next topic. This is one place where page tables would be useful, but there are others too (e.g., in fork).