Lecture 27: Virtual memory and paging in xv6

Mythili Vutukuru
IIT Bombay
https://www.cse.iitb.ac.in/~mythili/os/
Virtual address space in xv6

- 32-bit OS, so $2^{32}=4$GB virtual address space for every process
- Process address space divided into pages (4KB by default), and every valid logical page used by the process is mapped to a physical frame by the OS (no demand paging)
- Page table of a process maps virtual addresses to physical addresses
  - One page table entry per page, which contains the physical frame number (PFN) and various flags/permissions for the page
Page table in xv6

- Virtual address space = $2^{32}$ bytes, page size = 4KB = $2^{12}$ bytes
  - Up to $2^{20}$ pages per process
- Page table is a logical array of $2^{20}$ page table entries (PTE)
  - 20 bit page number is used to index into page table to locate PTE
- Each PTE has 20 bit physical frame number, and some flags
  - PTE_P indicates if page is present (if not set, access will cause page fault)
  - PTE_W indicates if writeable (if not set, only reading is permitted)
  - PTE_U indicates if user page (if not set, only kernel can access the page)
- Address translation: use page number (top 20 bits of virtual address) to index into page table, find physical frame number, add 12-bit offset
Two level page table

- $2^{20}$ PTEs cannot be stored contiguously, page table has two levels
  - $2^{10}$ “inner” page table pages, each with $2^{10}$ PTEs
  - Outer page directory stores PTE-like references to the $2^{10}$ inner page table pages
  - Physical address of outer page directory is stored in CPU’s cr3 register, used by MMU during address translation
- 32 bit virtual address = 10 bits index into page directory, next 10 bits index into inner page table, last 12 bits are offset within page
  - PFN from PTE + offset = physical address
Process virtual address space in xv6

- Memory image of a process starting at address 0 has
  - Code/data from executable
  - Fixed size stack (with guard page)
  - Expandable heap

- Kernel code/data is mapped beginning at address KERNBASE (2GB)
  - Kernel code/data
  - Free pages maintained by kernel
  - Some space reserved for I/O devices

- Page table of a process contains two sets of PTEs
  - User entries map low virtual addresses to physical memory used by the process for its code/data/stack/heap
  - Kernel entries map high virtual addresses to physical memory containing OS code and data structures (identical entries in all processes)

- Process can only access memory mapped by page table
  - Access only possible via virtual addresses in page table

- Different page table for every process, page table needs to be switched during a context switch
OS page table mappings (1)

- OS code/data structures part of virtual address space of every process.
  - Page table entries map high virtual addresses (2GB to 2GB+PHYSTOP) to OS code/data located in physical memory (0 to PHYSTOP)
  - Only one copy of OS code in memory, mapped into all process page tables
  - Kernel mappings are identical in all processes

- Can’t you directly access OS code using its physical address? No. With paging and MMU turned on, physical memory can only be accessed by assigning a virtual address to it, and adding a mapping from virtual to physical address in page table.

- What happens during a trap? The same page table can be used to access kernel during a trap. If OS is not part of virtual address space, will need new page table during trap.
OS page table mappings (2)

- Kernel page table mappings map virtual addresses 2GB: (2GB+PHYSTOP) to physical addresses 0 : PHYSTOP
  - 0 to PHYSTOP has memory for kernel code/data, I/O devices, mostly free pages

- Assigning free pages to processes
  - Suppose physical frame P is initially mapped into kernel part of address space at virtual address V (we will have V = P+2GB)
  - When assigned to a user process, P is assigned another virtual address U (<2GB)
  - Same frame P mapped twice into page table, at virtual addresses U and V
  - Kernel and user access same memory using different virtual addresses

- Every byte of RAM can consume 2 bytes of virtual address space, so xv6 cannot use more than 2GB of RAM (since max 32-bit virtual address space is 4GB)
  - Real kernels deal with this better
Maintaining free memory

• After boot up, RAM contains OS code/data and free pages
• OS collects all free pages into a free list, so that it can be assigned to user processes
  – Used for user memory (code/data/stack/heap) and page tables of user processes
• Free list is a linked list, pointer to next free page embedded within previous free page
  – Kernel maintains pointer to first page in the list
alloc and free operations

- Anyone who needs a free page calls `kalloc()`
  - Sets free list pointer to next page and returns first free page on list
- When memory needs to be freed up, `kfree()` is called
  - Add free page to head of free list, update free list pointer

```c
3186 char*
3187 kalloc(void)
3188 {
3189   struct run *r;
3190   if(kmem.use_lock)
3191     acquire(&kmem.lock);
3192   r = kmem.freelist;
3193   if(r)
3194     kmem.freelist = r->next;
3195   if(kmem.use_lock)
3196     release(&kmem.lock);
3197   return (char*)r;
3198 }
```

```c
3163 void
3164 kfree(char *v)
3165 {
3166   struct run *r;
3167   if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
3168     panic("kfree");
3169   if(kmem.use_lock)
3170     acquire(&kmem.lock);
3171   r = (struct run*)v;
3172   if(kmem.use_lock)
3173     release(&kmem.lock);
3174   r->next = kmem.freelist;
3175   kmem.freelist = r;
3176 }
```
Summary of virtual memory in xv6

• Only virtual addressing, no demand paging
• 4GB virtual address space for each process
• 2 tier page table: outer pgdir, inner page tables
• Process address space has:
  – User memory image at low virtual addresses (<2GB)
  – Kernel code/data mapped at high virtual addresses
• Kernel part of address space has OS code/data, memory for I/O devices, and free pages
  – Assigned to user processes as needed