TCSS 422: OPERATING SYSTEMS

Introduction to Paging

Wes J. Lloyd
Institute of Technology
University of Washington - Tacoma

OBJECTIVES

- Paging
- Address translation
- Paging questions

PAGING

- Split up address space of process into *fixed sized pieces* called **pages**
- Alternative to *variable sized pieces* (Segmentation) which suffers from significant fragmentation
- Physical memory is split up into an array of fixed-size slots called **page frames**.
- Each process has a **page table** which translates virtual addresses to physical addresses

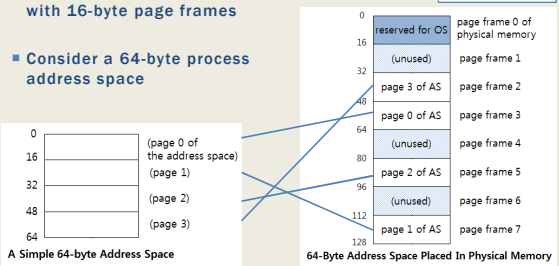
ADVANTAGES OF PAGING

- Flexibility
  - Abstracts the process address space into pages
  - No need to track direction of HEAP / STACK growth
  - No need to store unused space
- Simplicity
  - Pages and page frames are the same size
  - Easy to allocate and keep a free list of pages

PAGING: EXAMPLE

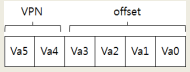
Page Table:
VP0 → PF3
VP1 → PF7
VP2 → PF5
VP3 → PF2

- Consider a 128 byte address space with 16-byte page frames
- Consider a 64-byte process address space

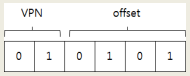


PAGING: ADDRESS TRANSLATION

- Two address components
  - VPN: Virtual Page Number
  - Offset: Offset within a Page



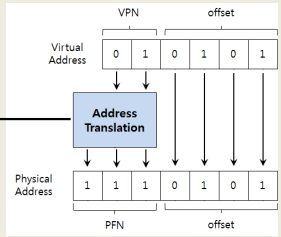
- Example:
  Page Size: 16-bytes, Address Space: 64-bytes



EXAMPLE: PAGING ADDRESS TRANSLATION

- Consider a 64-byte address space
- Stored in 128-byte physical memory
- Offset is preserved
- VPN is looked up

Page Table:
VP0 → PF3
VP1 → PF7
VP2 → PF5
VP3 → PF2



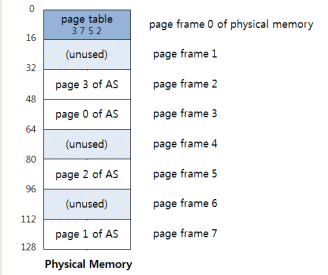
PAGING QUESTIONS

- Where are page tables stored?
- What are the typical contents of the page table?
- How big are page tables?
- Does paging make the system too slow?

WHERE ARE PAGE TABLES STORED?

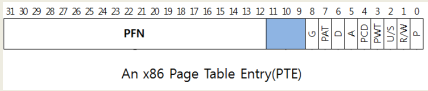
- Real world size example:
- Consider a 32-bit process address space (up to 4GB)
- With 4 KB pages
- 20 bits for VPN
- 12 bits for the page offset
- Page tables for each process are stored in RAM
  - Support potential storage of 2^{20} translations = 1,048,576 pages per process
  - Each page has a page table entry size of 4 bytes
- Consider 100+ OS processes
  - Requires 400+ MB of RAM to store process information

PAGE TABLE WITH KERNEL PHYSICAL MEMORY



WHAT'S ACTUALLY IN THE PAGE TABLE

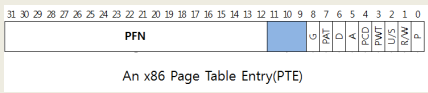
- Page table is data structure used to map virtual page numbers (VPN) to the physical address (Physical Frame Number PFN)
  - Linear page table → simple array
- Page-table entry
  - 32 bits for capturing state



An x86 Page Table Entry(PTE)

PAGE TABLE ENTRY

- P: present
- R/W: read/write bit
- U/S: supervisor
- A: accessed bit
- D: dirty bit
- PFN: the page frame number



An x86 Page Table Entry(PTE)

PAGE TABLE ENTRY - 2

- Common flags:
- **Valid Bit:** Indicating whether the particular translation is valid.
- **Protection Bit:** Indicating whether the page could be read from, written to, or executed from
- **Present Bit:** Indicating whether this page is in physical memory or on disk(swapped out)
- **Dirty Bit:** Indicating whether the page has been modified since it was brought into memory
- **Reference Bit(Accessed Bit):** Indicating that a page has been accessed

HOW BIG ARE PAGE TABLES?

- Page tables are too big to store on the CPU
- Page tables are stored using physical memory
- Paging supports efficiently storing a sparsely populated address space
  - Reduced memory requirement
    Compared to base and bounds, and segments

DOES PAGING MAKE THE SYSTEM TOO SLOW?

- Translation
- **Issue #1:** Starting location of the page table is needed
  - HW Support: Page-table base register
    - stores active process
    - Facilitates translation

Stored in RAM →

Page Table:
VP0 → PF3
VP1 → PF7
VP2 → PF5
VP3 → PF2

- **Issue #2:** Each memory address translation for paging requires an extra memory reference
  - HW Support: TLBs (Chapter 19)

PAGING MEMORY ACCESS

```
// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT

// Form the address of the page-table entry (PTE)
PTEAddr = PTBR + (VPN * sizeof(PTE))

// Fetch the PTE
PTE = AccessMemory(PTEAddr)

// Check if process can access the page
if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.ProtectBits) == False)
        RaiseException(PROTECTION_FAULT)
else
        // Access is OK: form physical address and fetch it
        offset = VirtualAddress & OFFSET_MASK
        PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
        Register = AccessMemory(PhysAddr)
```


COUNTING MEMORY ACCESSES

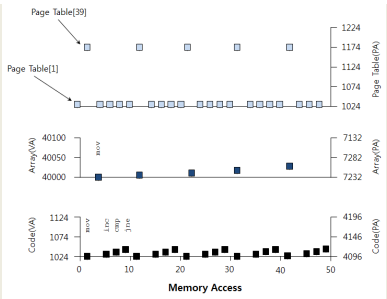
- Example: Use this Array initialization Code

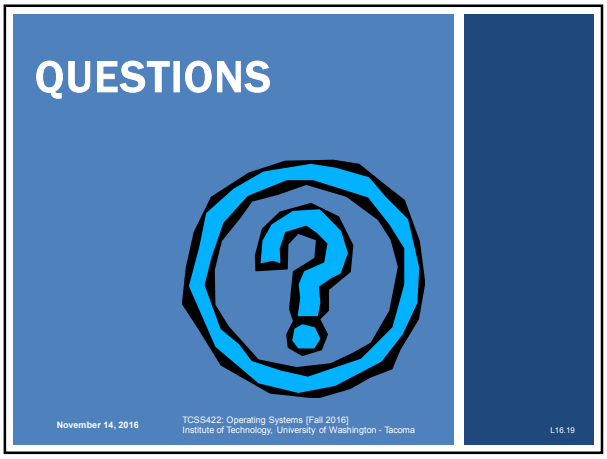
```
int array[1000];
...
for (i = 0; i < 1000; i++)
        array[i] = 0;
```

- Assembly equivalent:

```
0x1024 movl $0x0,(%edi,%eax,4)
0x1028 incl %eax
0x102c cmpl $0x03e8,%eax
0x1030 jne 0x1024
```


MEMORY ACCESSES: FOR THE FIRST 5 LOOP ITERATIONS




QUESTIONS
November 14, 2016
TCSS422: Operating Systems [Fall 2016]
Institute of Technology, University of Washington - Tacoma
L16.19