Design and Engineering of Computer Systems

Lecture 12: Paging

Mythili Vutukuru
IIT Bombay
Recap: Paging

- Virtual address space of a process is divided into **pages**, each page is stored in a free **physical memory frame** by OS
  - Mapping is remembered in the **page table**, one per process, part of PCB
- Virtual address space of a process has addresses for process code/data as well as shared software (language libraries, OS) that are used by process
  - One copy of OS in physical memory, mapped at high virtual addresses of all processes
- MMU uses page table to translate virtual addresses requested by CPU into physical addresses
  - Recent translations cached in **TLB**
- If address translation fails, MMU raises **trap**, CPU jumps to kernel mode
  - Otherwise, OS is not involved in address translation and memory access of user code
Structure of page table

- Array of page table entries, one per page of process
  - Each page table entry “points to” the physical frame corresponding to the page
- i-th page table entry (PTE) contains physical frame number and other details (permissions, status, ..) of i-th page of process
  - Valid: is this page in use by process (not all virtual addresses are used by process)
  - Read/write permissions
  - User/kernel permissions
  - Other status bits we will study later: present, dirty, accessed
- Address translation using page table
  - 32 bit virtual address = 20 bit page number + 12 bit offset
  - Use 20 bit page number to index into page table array
  - Find frame number for page number, add offset within page

\[ 4 \text{ KB} = 2^{12} \]

\[
\begin{align*}
\text{Page} \# & = \frac{1}{20} \\
\text{offset} & = \frac{1}{12}\text{ bits}
\end{align*}
\]

\[
\text{frame} \#
\]
Storing page table in memory

- What is typical size of page table?
  - 32-bit system, $2^{32} = 4\text{GB}$ virtual address space
  - Assume page size = $4\text{KB} = 2^{12}$
  - Number of pages = $(2^{32}/2^{12}) = 2^{20} = 1\text{M}$
  - Assume each page table entry is 4 bytes
  - Page table size of one process = 4MB

- How are page tables stored in memory?
  - All memory is only allocated in 4KB chunks

- Solution: split page table into pages (much like memory image), use another page table to keep track of original page table!
Hierarchical page table

• 4MB page table split into $2^{10}$ (1K) pages of 4KB each
• Physical frame numbers of $2^{10}$ pages containing “inner” page table stored in an outer page table or page directory
  • 4 byte page table entry each, so fits in one page of 4KB
• Page table has two levels
  • Outer page table (page directory) has physical frame numbers of $2^{10}$ “inner” page table pages
  • Each inner page table has physical frame numbers of $2^{10}$ pages of the process virtual address space
• MMU is given the physical address of the outer page directory
  • In case of TLB miss, uses 2 level page table to translate virtual address
Address translation in 2-level page table

- Virtual address of 32 bits = 20 bit page number + 12 bit offset
  - 20 bits = 10 bit index into page directory, 10 bit index into inner page table
  - Top most 10 bits to index into page directory, identify which one of $2^{10}$ inner page tables to use, next 10 bits index into inner page table, find one of $2^{10}$ page table entries, we now have the frame number of a page
  - Computer physical address using frame number and 12-bit offset into page

- MMU “walks” the page table to translate virtual addresses
Multi-level page tables

- More than two levels if outer page directory does not fit into one page
  - Store outer page directory in many pages, use yet another page table
  - This can go on until page table at a level fits in one page
- Example: 48-bit CPU, 4KB pages, 8 byte page table entries
  - \(2^{48}\) bytes in virtual address space = \(2^{36}\) pages for each process
  - Each page can store \(4KB/8 = 2^{9}\) page table entries
  - Innermost level has \(2^{36}\) page table entries = needs \(2^{27}\) pages
  - Next level has \(2^{18}\) pages, next level has \(2^{9}\) pages
  - Outermost level can store page table entries in 1 page
  - 4 level page table: 9 bits of virtual address to index into each level
- MMU page table walks become even longer, TLB hit rate is critical
  - MMU may have to perform 4 extra memory accesses before actual memory access!
Running a program

- User program is compiled into executable = program code + compile-time data (global/static variables)
- When program is run, OS creates process
  - Allocates new page table of process
  - Allocates physical frames to store executable code/data, stack, heap, ..
  - Builds virtual address space of process: adds mapping from virtual addresses (page numbers) to physical addresses (frame numbers) in page table
  - Pointer to page table provided to MMU when process is context switched in
  - CPU accesses virtual address, translated by MMU to physical address
- Virtual address space of a process has “gaps”
  - Some virtual addresses are not used for any code or data
  - Example: gap left between stack and heap to allow expansion
Heap management

- Heap: one or more pages of memory used for dynamic memory allocation at run time using malloc (and related functions)
  - Functions to allocate/deallocate from heap (malloc, free) provided by language libraries
- Heap manager gets memory from OS in page size chunks, allocates to user program in variable sized chunks
  - Memory allocation algorithm in the language library keeps track of all free chunks on the heap, finds the most suitable free chunk to return during malloc
  - Malloc returns the starting virtual address of the free chunk of requested size
  - Free chunks can be split or merged during allocation
  - Some language libraries automatically clean unused chunks, some do not (malloc-ed memory must be explicitly freed up by user, else memory leak)
- If no more free space in heap, heap manager asks OS for more memory via system call, obtains memory in page-sized chunks from OS
  - Can also return back memory to OS to shrink heap
System calls for memory allocation

• System calls to obtain page-sized memory from OS
  • The syscalls `brk/sbrk` are used to allocate page-sized chunks at the end of the data segment of executable (program break) where heap starts
  • `mmap` syscall used to obtain one or more page sized chunks at any unallocated range of virtual addresses
  • Of the two methods, mmap is more portable and preferable

• `mmap` system call for **memory mapping**
  • Takes size of memory required as argument (must be multiple of page size)
  • Returns the starting virtual address of the memory chunk allocated by OS
  • OS finds free physical memory frames, adds mapping from allocated virtual addresses to physical addresses
  • Allocated memory can be split into smaller chunks by heap manager
Custom memory allocation

- General-purpose malloc imposes performance overhead
  - Complex data structures to keep track of variable sized free chunks
- Some heaps optimized for fixed size allocation: slab allocators
  - Useful for user applications that allocate memory in fixed sizes
  - Heap memory is divided into fixed size chunks for allocation
  - More efficient than general-purpose variable sized allocation
- User programs can also directly call mmap to obtain one or more pages of memory from OS
  - Avoids using existing heap managers
  - Application can optimize data storage in the memory-mapped region
Summary

• In this lecture:
  • Page table structure ✓
  • Hierarchical paging ✓
  • System calls for memory management ✓

• Understand multi-level page tables: try to calculate page table sizes, number of levels of page tables for various values of virtual address spaces and page sizes

• Programming exercise: write a simple program to obtain one or more pages of memory via the mmap system call