Main Memory

1DV512 – Operating Systems

Dr. Kostiantyn Kucher

kostiantyn.kucher@lnu.se

November 18, 2020

Based on the Operating System Concepts slides by Silberschatz, Galvin, and Gagne (2018)

Suggested OSC book complement: Chapter 8

Agenda

Motivation

- Previously, we discussed multiprogramming and multitasking approaches $\Rightarrow$ by support multiple processes, we improve the efficiency of computer systems usage
- However, we must keep many processes in memory $\Rightarrow$ OS must manage the memory available to the processes!
- Program must be brought (from disk) into memory and placed within a process for it to be run
- CPU can only access directly its registers and *main memory* (RAM)
- *Memory management unit* (MMU) only sees a stream of:
  - addresses + read requests, and
  - addresses + data + write requests
- Register access is done in one CPU clock (or less)
- Main memory can take many cycles, causing a *stall* $\Rightarrow$ additional fast *cache* is added between CPU registers and main memory
- Besides the performance, *protection* of memory is also a concern $\Rightarrow$ user processes vs the OS, user processes vs each other...

Basic Memory Protection

- OS doesn't usually intervene between the CPU and its memory accesses for performance reasons ⇒ some level of hardware support is necessary!
- Basic protection approach ⇒ make sure that each process has a *separate memory space*
- Typical solution ⇒ a pair of *base* and *limit* registers that define the *logical address space* of a process
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user
- The instructions to loading the base and limit registers are privileged ⇒ kernel-mode only!

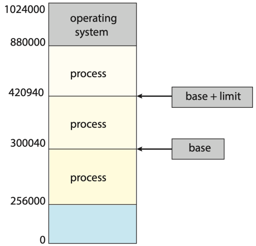


Fig. 9.1 in OSC book

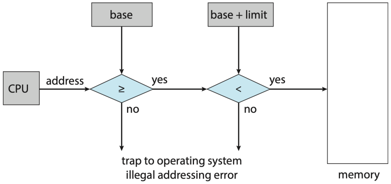


Fig. 9.2 in OSC book

Address Binding

- Usually, a program resides on a disk as a binary executable file ⇒ must be loaded into the memory of a process to be executed!
- As the process executes, it accesses instructions and data from memory at certain *addresses*
- *Address binding* of instructions and data to memory addresses can happen at three different stages:
  - *Compile time* ⇒ if memory location known a priori, *absolute code* can be generated
  - *Load time* ⇒ must generate *relocatable code* if memory location is not known at compile time
  - *Execution time* ⇒ binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)
    - Most operating systems use this method!

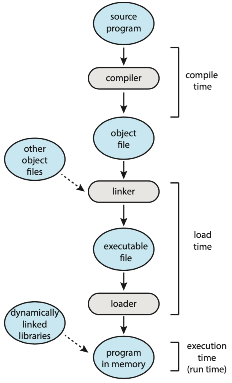


Fig. 9.3 in OSC book

Logical and Physical Addresses

- *Logical address* ⇒ generated by the CPU; also referred to as *virtual address*
- *Physical address* ⇒ address seen by the memory management unit (MMU)
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes, and different in the execution-time address binding
- One simple approach for execution-time *relocation* for MMU ⇒ add the value from the *relocation register* to each logical address
- The user program deals with logical addresses (in range 0 .. *max*) ⇒ it never sees the real physical addresses (in range $R + 0$ .. $R + max$)!
- Thus, a logical address space is bound to a separate physical address

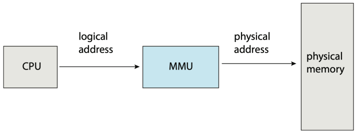


Fig. 9.4 in OSC book

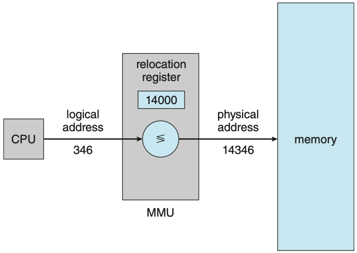


Fig. 9.5 in OSC book

Dynamic Loading and Dynamic Linking

- *Dynamic loading* ⇒ instead of loading the complete program in memory, only the main necessary part (routine) is loaded
- All routines kept on disk in relocatable load format and loaded as they are invoked for the first time
- No special support from the OS is required ⇒ however, OS can assist by providing libraries/tools to implement this

- *Dynamic linking* ⇒ binding delayed until the execution time, so that some libraries could be *shared* (another term: *dynamically loaded libraries*, DLLs)
- When a corresponding routine is invoked, the OS loads the library into memory (if necessary) and adjusts the addresses in the calling program
- Dynamic linking and shared libraries typically require support from the OS

Agenda

- Motivation and Introduction
- Contiguous Memory Allocation
- Paging
- Swapping
- Memory Management Examples
- Summary

Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource ⇒ must be allocate efficiently!
- *Contiguous allocation* is one early and straightfotward method
- Main memory usually divided into two partitions:
  - Resident operating system ⇒ usually held in high memory with interrupt vector
  - User processes then held in low memory ⇒ each process contained in single *contiguous* section of memory
  - Note: placing OS in high memory is a design choice, and low memory could be used instead in some cases

Memory Protection

- Relocation register + limit register can be used to protect user processes from each other, and from changing operating-system code and data
- Base register contains the value of the smallest physical address
- Limit register contains the range of logical addresses ⇒ each logical address must be less than the limit register
- MMU maps logical address dynamically ⇒ can allow actions such as kernel code being transient and kernel changing size

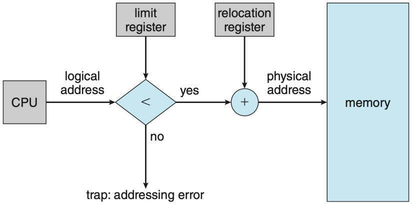


Fig. 9.6 in OSC book

Memory Allocation and Variable Partitions

- *Variable-partition allocation* ⇒ assign a memory partition to each process according to its needs
- *Hole* ⇒ a block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- An exiting process frees its partition, and adjacent free partitions are combined
- Operating system maintains information about: a) allocated partitions and b) free partitions (holes)

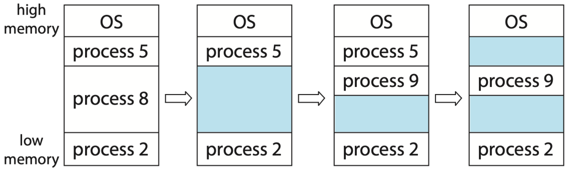


Fig. 9.7 in OSC book

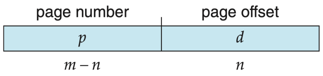
Dynamic Storage Allocation Problem and Fragmentation

- How to satisfy a request of size *n* from the list of free holes?
  - *First fit* ⇒ allocate the first hole that is big enough
  - *Best fit* ⇒ allocate the smallest hole that is big enough; must search entire list, unless it is ordered by size
  - *Worst fit* ⇒ allocate the largest hole; must also search entire list, unless it is ordered by size
- According to simulations, *first fit* and *best fit* are both good options with regard to storage utilization; and *first fit* is generally faster
- Both strategies suffer from *external fragmentation* ⇒ enough total memory space exists to satisfy the next request, but it is not contiguous!
  - *First fit* analysis reveals that for N allocated blocks, $0.5N$ is lost to fragmentation ⇒ *50-percent rule*
  - Can be addressed by *compaction* (not always possible) or *paging* ⇒ discussed later!
- *Internal fragmentation* ⇒ allocated memory (multiple fixed-size blocks) may be slightly larger than requested memory

Agenda

Paging — Basic Method

- *Paging* $\Rightarrow$ a memory management scheme that permits a process's physical address space to be non-contiguous
- Used in most modern OSs and implemented with both software (OS) and hardware support
- Prevents external fragmentation, although internal fragmentation is still possible
- Basic method:
  - Divide physical memory into fixed-sized blocks called *frames*
  - Divide logical memory into blocks of same size called *pages*
  - Frame size $\Rightarrow$ usually power of 2, between 512 bytes and 16 MB
  - Logical address space can now be much larger than the amount of RAM!
  - For a program with a size of N pages $\Rightarrow$ N free frames required in total
  - Every address generated by the CPU is divided into two parts: a *page number* (p) and a *page offset* (d) for given logical address space of size 2^m and page size 2^n



Sec. 9.3.1 in OSC book

Page Table

- A *page table* is used to translate logical to physical addresses for a process
- It contains the base address of each frame in physical memory
- Thus, a page number *p* and a page offset *d* are used with the page table
- MMU replaces *p* in the logical address with the corresponding frame number *f*; *d* is not changed

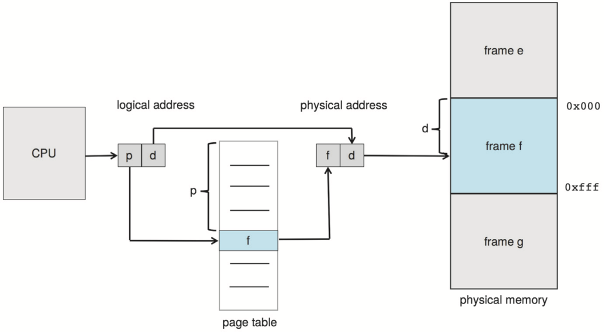


Fig. 9.8 in OSC book

Paging Model of Logical and Physical Memory

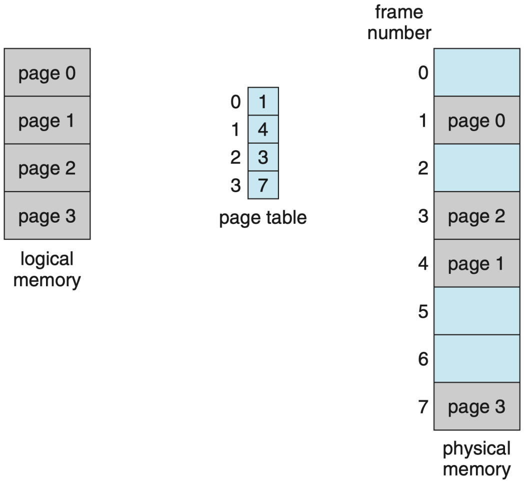


Fig. 9.9 in OSC book

Paging Example

- Here, $n = 2$ bits $\Rightarrow$ page size is $2^n = 4$ bytes
- $m = 4$ bits $\Rightarrow$ logical address space size is $2^m = 16$ bytes
- Available physical memory is 32 bytes $\Rightarrow$ 8 pages

- For logical address 0 $\Rightarrow p = 0, d = 0$ $\Rightarrow f = 5 \Rightarrow$ physical address is $f \cdot 2^n + d = 5 \cdot 4 + 0 = 20$
- For logical address 13 $\Rightarrow p = 3, d = 1 \Rightarrow f = 2$ $\Rightarrow$ physical address is $f \cdot 2^n + d = 2 \cdot 4 + 1 = 9$

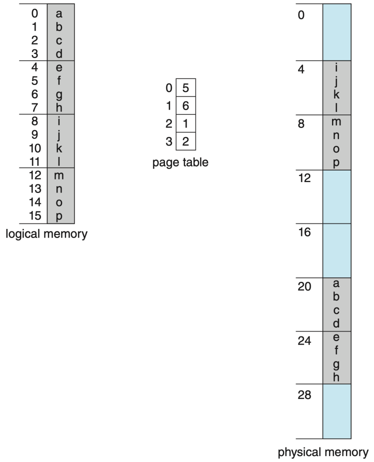


Fig. 9.10 in OSC book

Free Frames Before and After Process Allocation

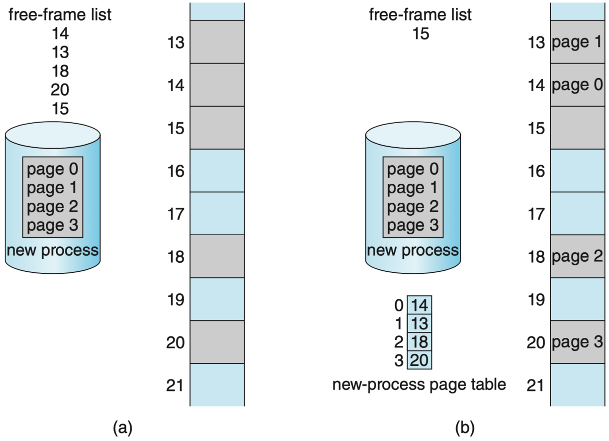


Fig. 9.11 in OSC book

Hardware Support

- Page tables are *per-process* data structures $\Rightarrow$ stored in PCB for each process
- Hardware support approaches for improving page table processing performance:
  - Dedicated high-speed CPU registers $\Rightarrow$ efficient address translation, but increased context switching time + not feasible for thousands of pages!
  - *Page-table base register (PTBR)* $\Rightarrow$ keep a pointer to respective the page table stored in main memory
  - *Translation look-aside buffer (TLB)* $\Rightarrow$ small and high-speed hardware cache for recent look-ups

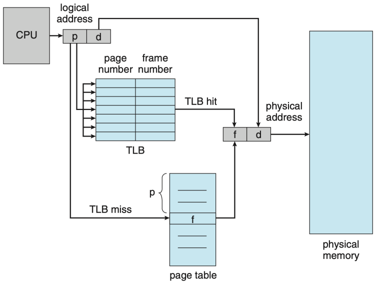


Fig. 9.12 in OSC book

Memory Protection

- Memory protection implemented by associating protection bit with each frame $\Rightarrow$ indicate if *read-only* or *read-write* access is allowed
- Can also add more bits to indicate page *execute-only*, and so on
- *Valid-invalid* bit attached to each entry in the page table:
  - *valid* $\Rightarrow$ the page is in the process logical address space, and is thus a legal page
  - *invalid* $\Rightarrow$ the page is not in the process logical address space
  - Alternatively, use *page-table length register (PTLR)*
- Any violations result in a trap to the kernel

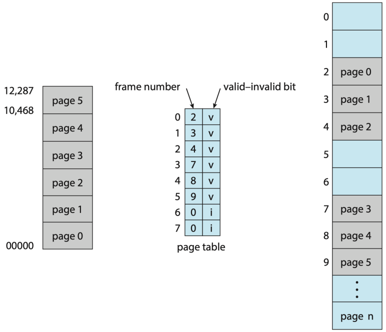


Fig. 9.13 in OSC book

Shared Pages

- *Shared code* ⇒ only one copy of read-only (*reentrant*) code shared among processes
- Efficient memory space usage ⇒ e.g., for widely used shared libraries such as standard C library
- Also useful for IPC if sharing of read-write pages is allowed

- Each process also keeps its own private code and data
- The pages for the private code and data can appear anywhere in the logical address space

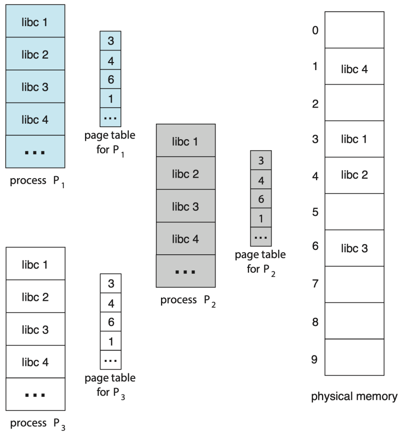


Fig. 9.14 in OSC book

Advanced Page Table Structures

- Memory structures for paging can get huge using basic, straightforward methods ⇒ millions of page table entries, megabytes of memory for each process for the page table itself. . .
- Thus, several more advanced approaches are applied ⇒ the idea is to divide the page table into smaller units
- *Hierarchical Paging* ⇒ multiple page tables (e.g., two-level) + paging the page table (so-called *forward-mapped page table*)
- *Hashed Page Tables* ⇒ for >32-bit systems, more efficient to use a virtual page number hashed into a page table
- *Inverted Page Tables* ⇒ keep only a single page table and track all physical pages/frames used by processes

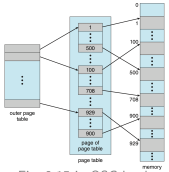


Fig. 9.15 in OSC book

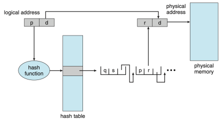


Fig. 9.17 in OSC book

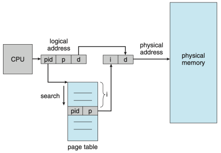


Fig. 9.18 in OSC book

Agenda

Swapping

- If the amount of physical memory is insufficient ⇒ a process can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory for continued execution
- *Backing store* ⇒ fast disk large enough to accommodate copies of all memory images for all users
- Standard swapping was used in traditional Unix systems, but is no longer used in modern OSs ⇒ too slow to swap complete processes!
- E.g., mobile OSs (iOS, Android) remove process data if free memory is low
- Most modern systems support *swapping with paging* ⇒ the *page out* operation moves a page to the backing store, and the *page in* operation does the opposite

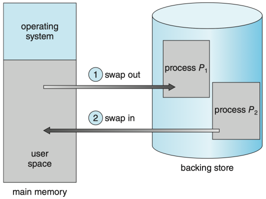


Fig. 9.19 in OSC book

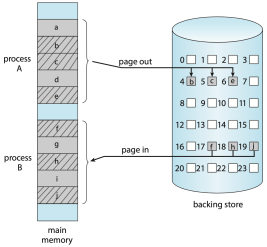


Fig. 9.20 in OSC book

Agenda

Intel 32- and 64-bit Architectures

- *IA-32* involves *segmentation* + paging ⇒ up to 16000 segments per process, each segment up to 4 GB ⇒ based on the logical address, a 32-bit *linear address* is generated
- Logical address space for each process divided into *local* (process-private) and *shared* partitions
- Two-level paging involving either 4 KB or 4 MB pages
- *Page Address Extension (PAE)* used to address >4 GB ⇒ using a three-level scheme leads to the support of up to 64 GB of RAM, if supported by OS
- *x86-64* supports up to 64-bit address space ⇒ 48 bits are currently used in practice, with a four-level paging scheme

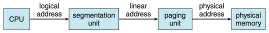


Fig. 9.21 in OSC book

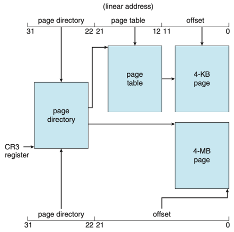


Fig. 9.23 in OSC book

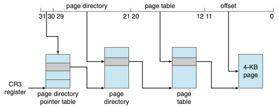


Fig. 9.24 in OSC book

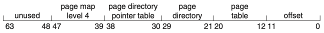


Fig. 9.25 in OSC book

ARM 64-bit Architecture

- *ARMv8* supports three different *translation granules*: 4 KB, 16 KB, and 64 KB
- Each translation granule provides different page sizes, as well as larger sections of contiguous memory, known as *regions*
- 4 KB and 16 KB granules ⇒ up to four levels of paging; 64 KB ⇒ up to three
- 48 (out of 64) bits currently used for addresses
- *Translation Table Base Register (TTBR)* ⇒ points to table level 0 for the current thread
- ARM supports two levels of translation look-aside buffers (TLBs) ⇒ two *micro TLBs* (instructions + data) at inner level and one *main TLB* at outer level

Translation Granule Size	Page Size	Region Size
4 KB	4 KB	2 MB, 1 GB
16 KB	16 KB	32 MB
64 KB	64 KB	512 MB

Sec. 9.7 in OSC book

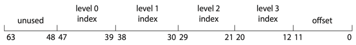


Fig. 9.26 in OSC book

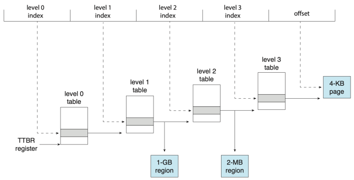


Fig. 9.27 in OSC book

Linnæus University

Agenda

Summary

- One way to allocate memory address space to each process is through the use of base and limit registers
- CPU generates a logical address, and MMU translates it to a physical address
- Modern OSs typically use memory paging with the page size of either 4 KB or 8 KB, and additionally support *huge pages* for special scenarios
- 64-bit and even 32-bit architectures often require advanced paging techniques involving multiple paging levels and hashing