Operating Systems
[9. Main Memory]

Chung-Wei Lin

cwlin@csie.ntu.edu.tw

CSIE Department

National Taiwan University

From Operating System Concepts (10th Edition)

Objectives

- Explain the difference between a logical and a physical address and the role of the memory management unit (MMU) in translating addresses
- Apply first-, best-, and worst-fit strategies for allocating memory contiguously
- Explain the distinction between internal and external fragmentation
- Translate logical to physical addresses in a paging system that includes a translation look-aside buffer (TLB)
- Describe hierarchical paging, hashed paging, and inverted page tables

Outline

- **Background**
  - Basic Hardware
  - Address Binding
  - Logical Versus Physical Address Space
  - Dynamic Loading
  - Dynamic Linking and Shared Libraries
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Example: Intel 32- and 64-bit Architecture
- Example: ARMv8 Architecture

Background

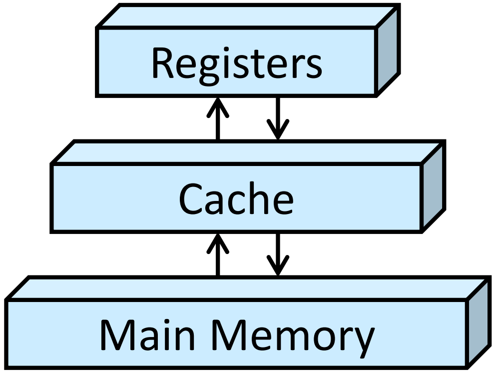
- A typical instruction-execution cycle
  - Fetch an instruction from memory
  - Decode the instruction
  - Fetch operands from memory
  - Execute the instruction
  - Store back results in memory
- We are interested only in the sequence of memory addresses generated by the running program
  - The memory unit sees only a stream of memory addresses
  - It does not know how they are generated or what they are for (instructions or data)

Outline

- **[Background]**
  - **Basic Hardware**
  - Address Binding
  - Logical Versus Physical Address Space
  - Dynamic Loading
  - Dynamic Linking and Shared Libraries
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Example: Intel 32- and 64-bit Architecture
- Example: ARMv8 Architecture

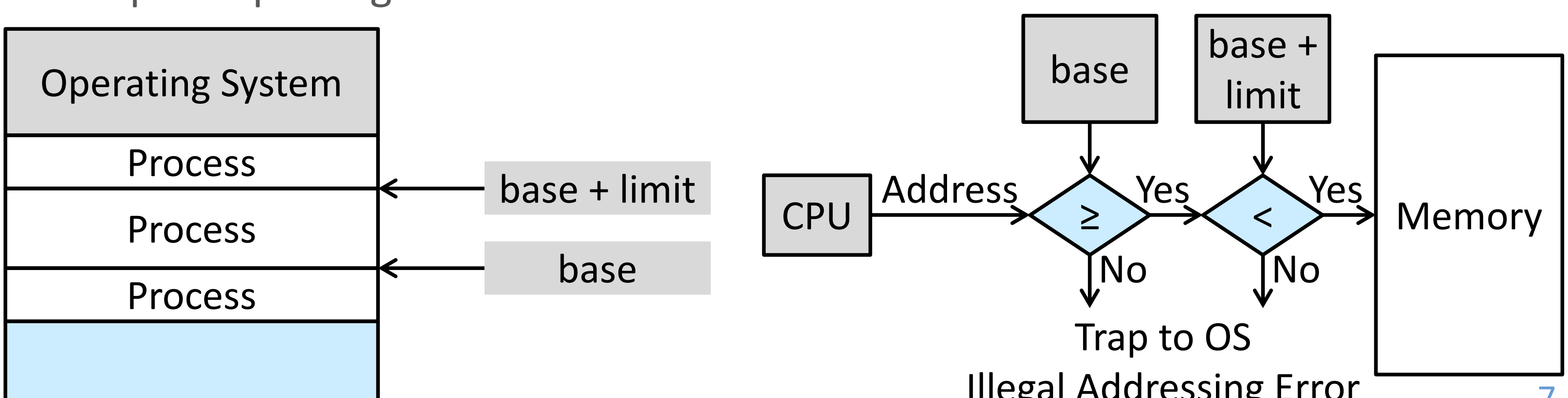
Basic Hardware

- The general-purpose storage that the CPU can access directly
  - Main memory
  - Registers built into each CPU core
- **Cache** is also added between the CPU and main memory
  - Reason
    - Registers built into each CPU core are generally accessible within one cycle of the CPU clock
    - Completing a memory access may take many cycles of the CPU clock, and thus the CPU may need to **stall**



Address Protection

- This protection must be provided by the hardware
  - An operating system does not usually intervene between the CPU and its memory accesses
- The protection compares every address generated in user mode with two registers
  - The **base register** holds the smallest legal physical memory address
  - The **limit register** specifies the size of the range
  - The registers can be loaded only by the operating system which uses a special privileged instruction

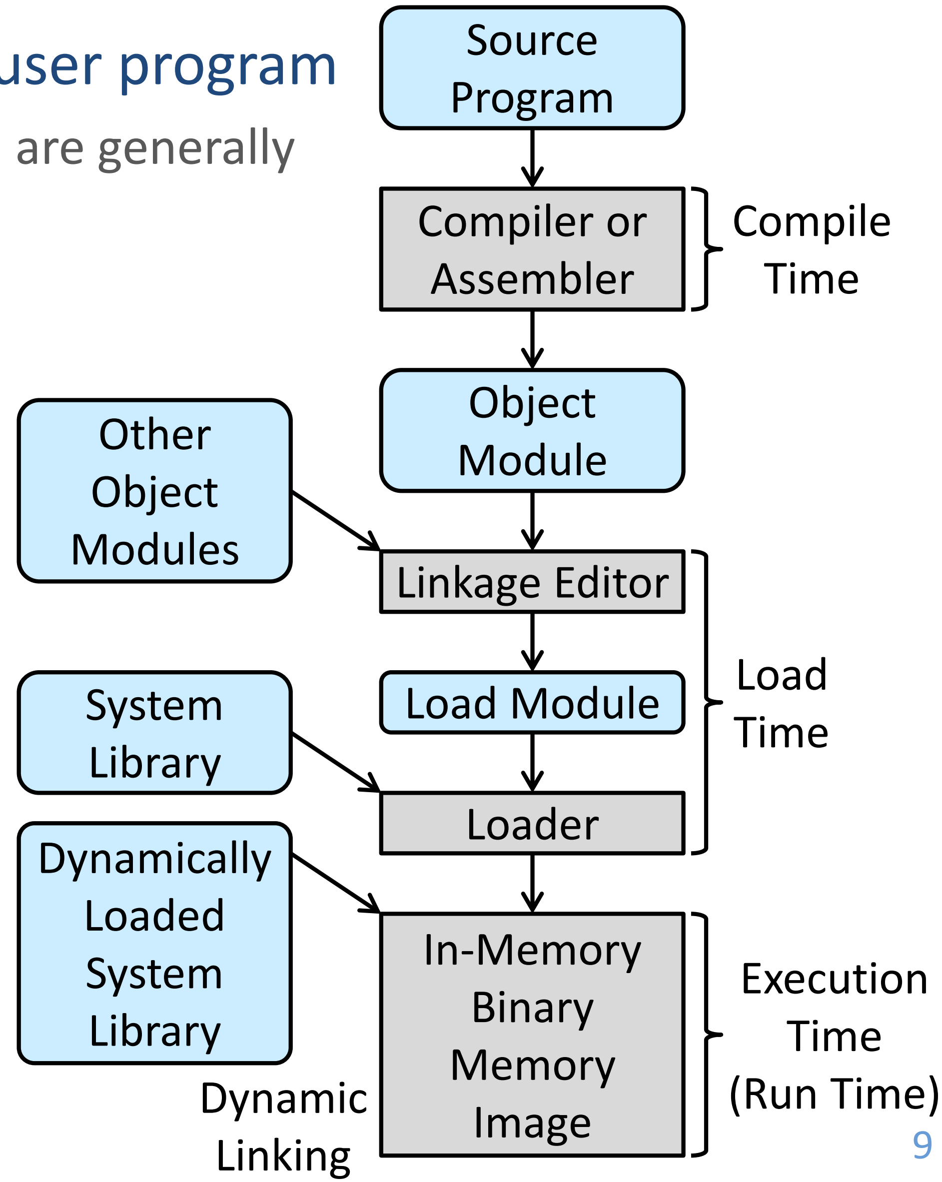


Outline

- **Background**
  - Basic Hardware
  - **Address Binding**
  - Logical Versus Physical Address Space
  - Dynamic Loading
  - Dynamic Linking and Shared Libraries
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Example: Intel 32- and 64-bit Architecture
- Example: ARMv8 Architecture

Address Binding (1/2)

- The steps before executing a user program
  - Addresses in the source program are generally symbolic
    - Example: a variable `count`
  - A compiler typically **binds** the symbolic addresses to relocatable addresses
    - Example: 14 bytes from the beginning of this module
  - A linker or loader binds the relocatable addresses to absolute addresses
    - Example: 74014
  - Each binding is a mapping from one address space to another



Address Binding (2/2)

- The binding of instructions and data to memory addresses at
  - Compile time
    - If the location of a process in memory is known, then the compiler can generate **absolute code**
    - If the starting location changes, then recompiling is needed
  - Load time
    - If the location of a process in memory is not known at compile time, then the compiler generates **relocatable code**
    - The final binding is delayed until load time
    - If the starting address changes, the reloading is needed
  - Execution time
    - If the process can be moved within memory during its execution, then the binding is delayed until run time
    - Special hardware must be available (next slide)
    - Most operating systems use this method

Outline

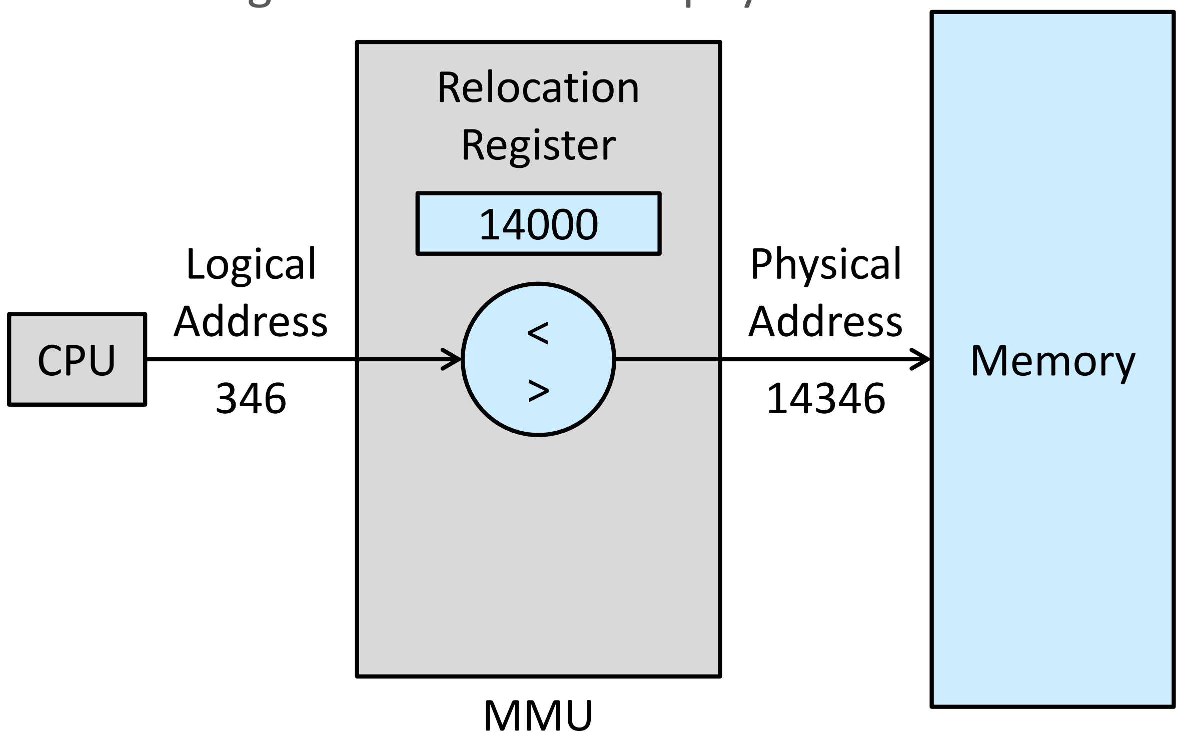
- ❑ **[Background]**
  - ➢ Basic Hardware
  - ➢ Address Binding
  - ➢ **Logical Versus Physical Address Space**
  - ➢ Dynamic Loading
  - ➢ Dynamic Linking and Shared Libraries
- ❑ Contiguous Memory Allocation
- ❑ Paging
- ❑ Structure of the Page Table
- ❑ Swapping
- ❑ Example: Intel 32- and 64-bit Architecture
- ❑ Example: ARMv8 Architecture

Logical Versus Physical Address Space

- **Logical address**
  - An address generated by the CPU
- **Physical address**
  - An address seen by the memory unit, i.e., the one loaded into the **memory-address register** of the memory
- Binding addresses has
  - Identical logical and physical addresses at either compile or load time
  - Different logical and physical addresses at execution time
    - In this case, we usually refer to the logical address as a **virtual address**
- **Logical address space**
  - Set of all logical addresses generated by a program
- **Physical address space**
  - Set of all physical addresses corresponding to these logical addresses

Memory-Management Unit (MMU)

- Hardware device performing run-time mapping from virtual to physical addresses
  - The base register is now called a **<u>relocation register</u>**
  - The user program deals with logical addresses
  - MMU converts logical addresses into physical addresses



Outline

- ❑ **Background**
  - ➢ Basic Hardware
  - ➢ Address Binding
  - ➢ Logical Versus Physical Address Space
  - ➢ **Dynamic Loading**
  - ➢ Dynamic Linking and Shared Libraries
- ❑ Contiguous Memory Allocation
- ❑ Paging
- ❑ Structure of the Page Table
- ❑ Swapping
- ❑ Example: Intel 32- and 64-bit Architecture
- ❑ Example: ARMv8 Architecture

Dynamic Loading

- A routine is not loaded until it is called for better memory-space utilization
  - All routines are kept on disk in a relocatable load format
  - The main program is first loaded into memory and executed
- This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases
  - Examples: error routines
- Dynamic loading does not require special support from the operating system
  - It is the responsibility of the users to design their programs to take advantage of such a method
  - Operating systems may help the programmer by providing library routines to implement dynamic loading

Outline

- **Background**
  - Basic Hardware
  - Address Binding
  - Logical Versus Physical Address Space
  - Dynamic Loading
  - **Dynamic Linking and Shared Libraries**
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Example: Intel 32- and 64-bit Architecture
- Example: ARMv8 Architecture

Dynamic Linking and Shared Libraries

- Static linking
  - System libraries are treated like any other object module
- Dynamic linking
  - Linking is postponed until execution time
    - Advantages: decrease the size of an executable image (save main memory), enable sharing among multiple processes, and enable updates
- **Dynamically linked libraries** (DLLs) are system libraries that are linked to user programs when programs are run
  - Only one instance of the DLL is in main memory
    - For this reason, DLLs are also known as **shared libraries**
  - DLLs are used extensively in Windows and Linux systems
- Dynamic linking and shared libraries generally require support from the operating system

Outline

- ❑ Background
- ❑ **Contiguous Memory Allocation**
  - ➢ Memory Protection
  - ➢ Memory Allocation
  - ➢ Fragmentation
- ❑ Paging
- ❑ Structure of the Page Table
- ❑ Swapping
- ❑ Example: Intel 32- and 64-bit Architecture
- ❑ Example: ARMv8 Architecture

Contiguous Memory Allocation

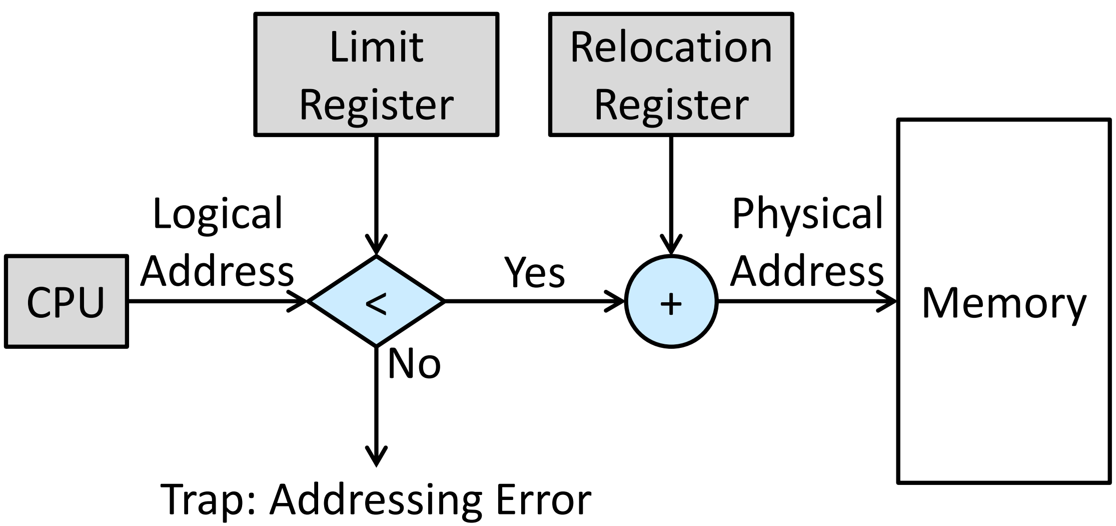
- The memory is usually divided into two partitions
  - One for the operating system
    - Many operating systems (including Linux and Windows) place the operating system in high memory
  - One for the user processes
- **Contiguous memory allocation** (one early method)
  - Each process is contained in a single section of memory that is contiguous to the section containing the next process

Outline

- Background
- **Contiguous Memory Allocation**
  - **Memory Protection**
  - Memory Allocation
  - Fragmentation
- Paging
- Structure of the Page Table
- Swapping
- Example: Intel 32- and 64-bit Architecture
- Example: ARMv8 Architecture

Memory Protection

- Combine two ideas previously discussed
  - The limit register contains the range of logical addresses
  - The relocation register contains the value of the smallest physical address
- The dispatcher loads the relocation and limit registers with the correct values as part of the context switch

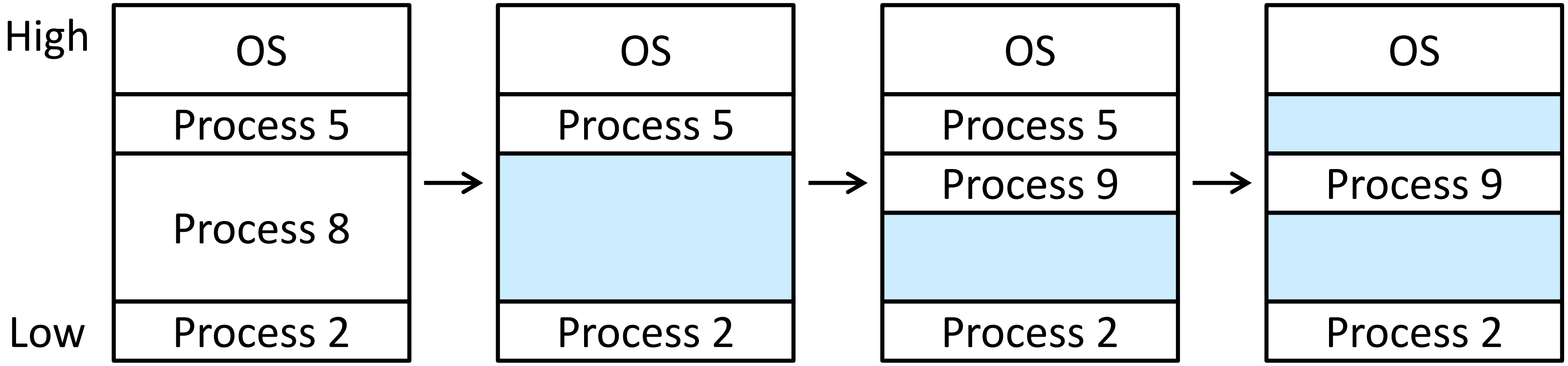


Outline

- Background
- **Contiguous Memory Allocation**
  - Memory Protection
  - **Memory Allocation**
  - Fragmentation
- Paging
- Structure of the Page Table
- Swapping
- Example: Intel 32- and 64-bit Architecture
- Example: ARMv8 Architecture

Memory Allocation

- Assign processes to variably sized partitions in memory, where each partition may contain exactly one process
  - A **variable-partition** scheme keeps a table indicating which parts of memory are available and which are occupied
  - Initially, all memory is available and considered one large block, a **hole**
  - When a process arrives, it is
    - Allocated space and loaded into memory
    - Rejected and provided an error message, or placed into a wait queue
  - When a process terminates, it releases its memory
    - Adjacent holes are merged



Dynamic Storage Allocation Problem

- How to satisfy a request of size n from a list of free holes
  - **First fit** allocates the first hole that is big enough
  - **Best fit** allocates the smallest hole that is big enough
    - Search the entire list, unless the list is ordered by size
    - Produce the smallest leftover hole
  - **Worst fit** allocates the largest hole
    - Search the entire list, unless the list is ordered by size
    - Produce the largest leftover hole
  - Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization

Outline

- Background
- **Contiguous Memory Allocation**
  - Memory Protection
  - Memory Allocation
  - **Fragmentation**
- Paging
- Structure of the Page Table
- Swapping
- Example: Intel 32- and 64-bit Architecture
- Example: ARMv8 Architecture

Internal Fragmentation

- Example
  - A hole of 18,464 bytes
  - A process requests 18,462 bytes
  - If we allocate exactly the requested block, a hole of 2 bytes is left
    - The overhead to keep track of this hole will be substantially larger than the hole itself
- **Internal fragmentation**
  - Break the physical memory into fixed-sized blocks and allocate memory in units based on block size, to avoid the problem above
    - The memory allocated to a process may be slightly larger than the requested memory
    - The difference between these two numbers is internal fragmentation (unused memory that is internal to a partition)

External Fragmentation

- **External fragmentation**
  - There is enough total memory space to satisfy a request but the available spaces are not contiguous
    - Storage is fragmented into a large number of small holes
  - 50-percent rule
    - Statistical analysis reveals that, even with some optimization, given N allocated blocks, another 0.5N blocks will be lost to fragmentation
- Solutions
  - **Compaction** with dynamic relocation at execution time
    - Shuffle the memory contents to place all free memory together
  - **Paging** (next section)
    - Permit the "physical" address space of processes to be noncontiguous
    - The most common memory-management technique for computer systems

Outline

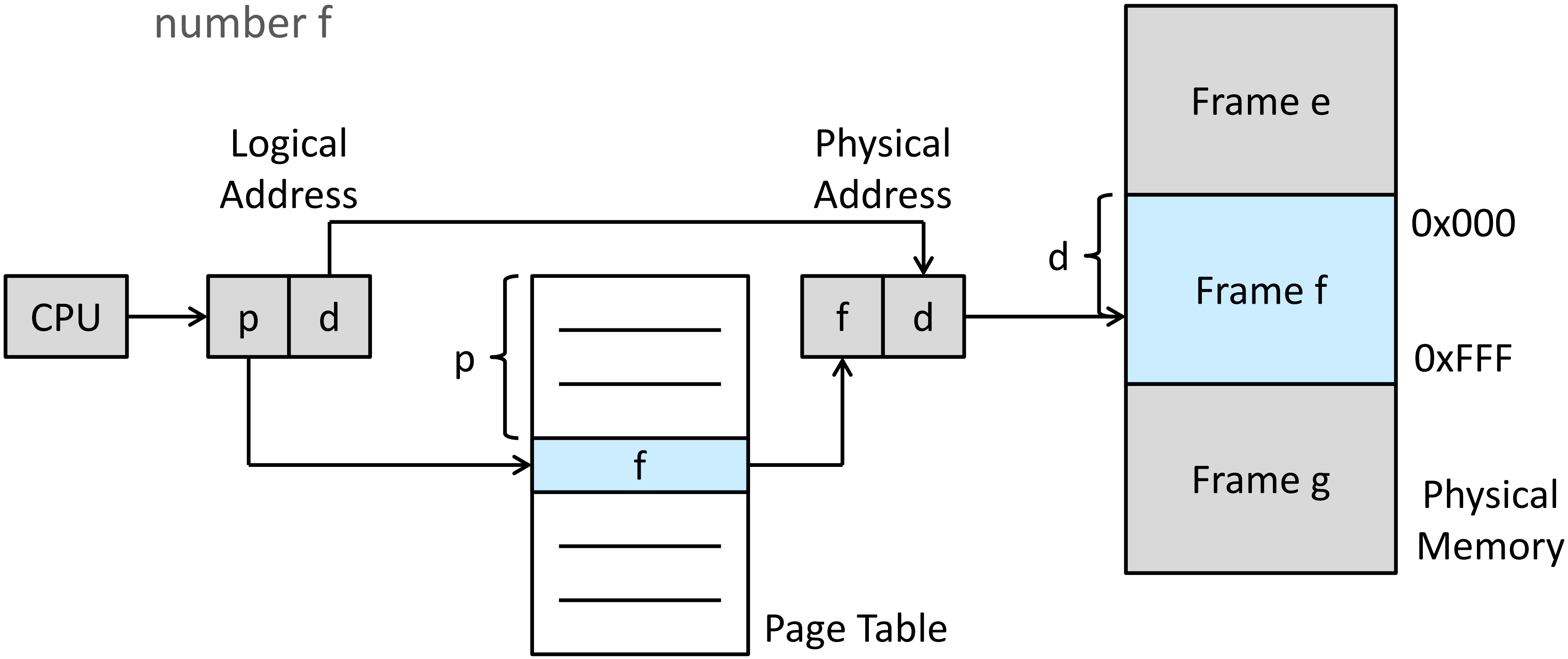
- Background
- Contiguous Memory Allocation
- <u>**Paging**</u>
  - <u>**Basic Method**</u>
  - Hardware Support
  - Protection
  - Shared Pages
- Structure of the Page Table
- Swapping
- Example: Intel 32- and 64-bit Architecture
- Example: ARMv8 Architecture

Pages and Frames

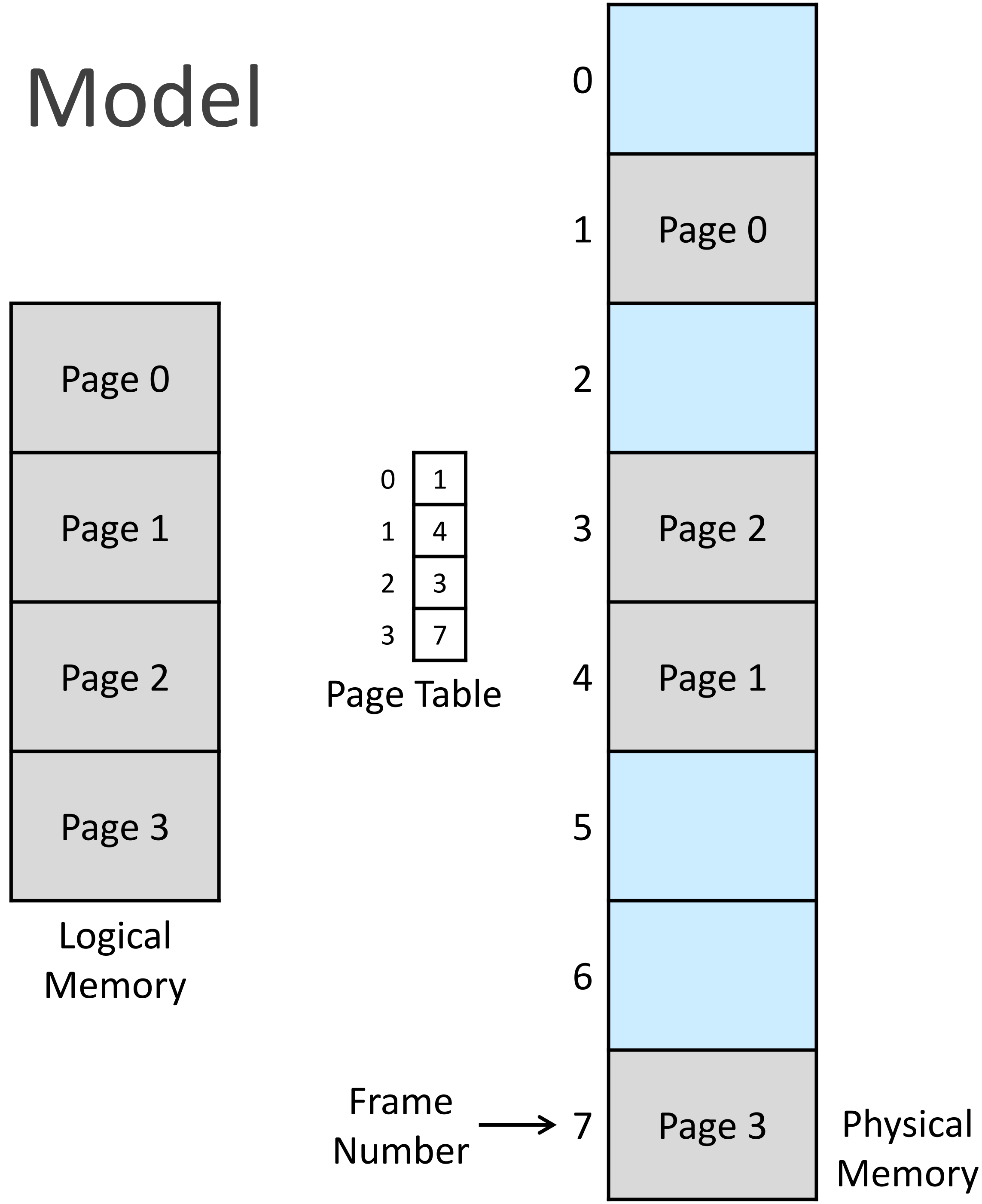
- When a process is to be executed, its **pages** are loaded into any available memory **frames** from their source
  - Pages: fixed-sized blocks of logical memory
  - Frames: fixed-sized blocks of physical memory
- Every address generated by the CPU is divided into two parts
  - A **page number** p: an index into a **per-process** **page table**
  - A **page offset** d: the location in the frame being referenced
- The page size (= the frame size) is defined by the hardware
  - The size is a power of 2, typically between 4 KB and 1 GB
  - If the size of the logical address space is 2^m, and a page size is 2^n bytes
    - The high-order $(m - n)$ bits of a logical address designate the page number
    - The low-order n bits of a logical address designate the page offset

Paging Hardware

- Steps taken by the MMU to translate a logical address generated by the CPU to a physical address
  - Extract the page number p and use it as an index into the page table
  - Extract the corresponding frame number f from the page table
  - Replace the page number p in the logical address with the frame number f

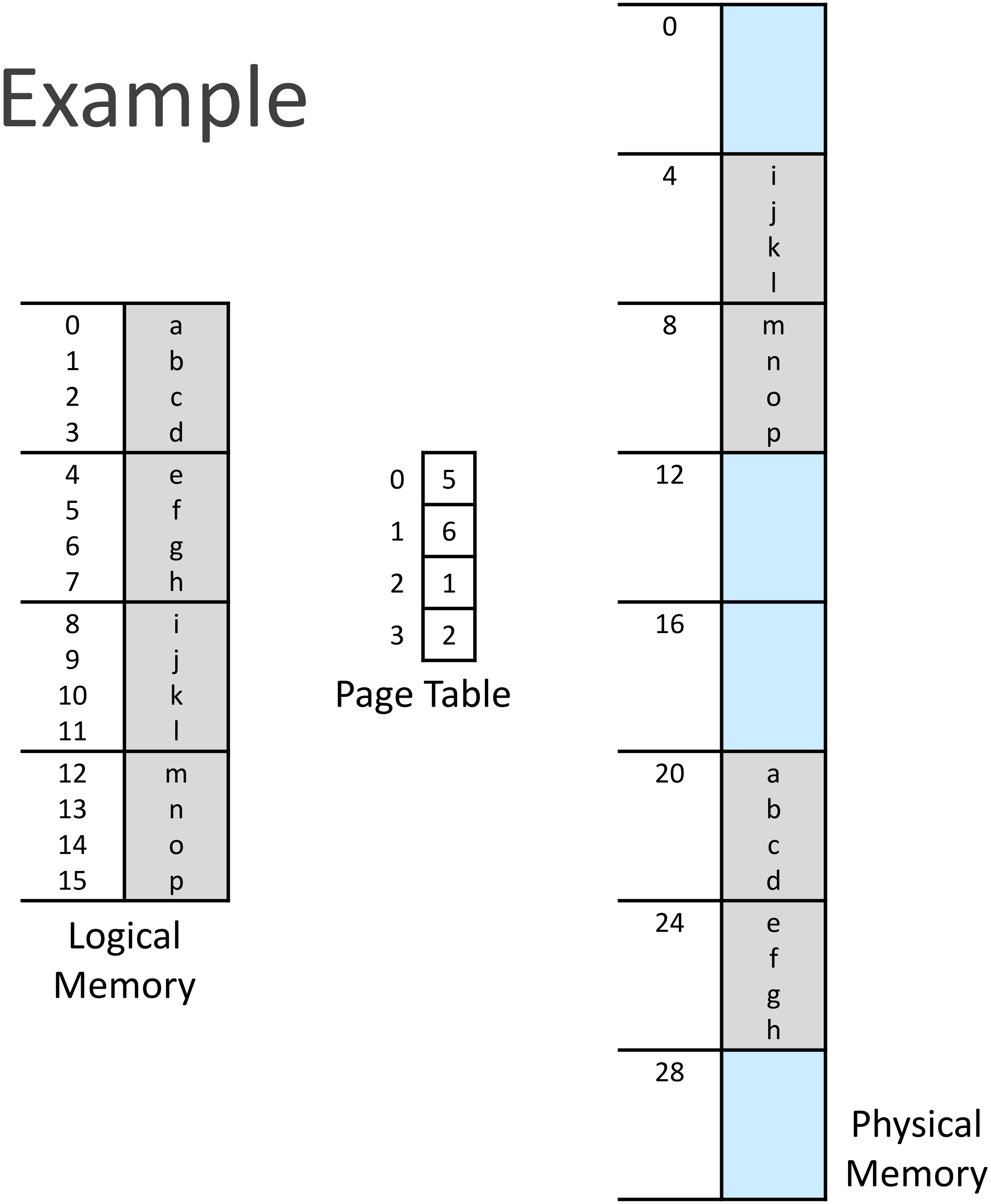


Paging Model

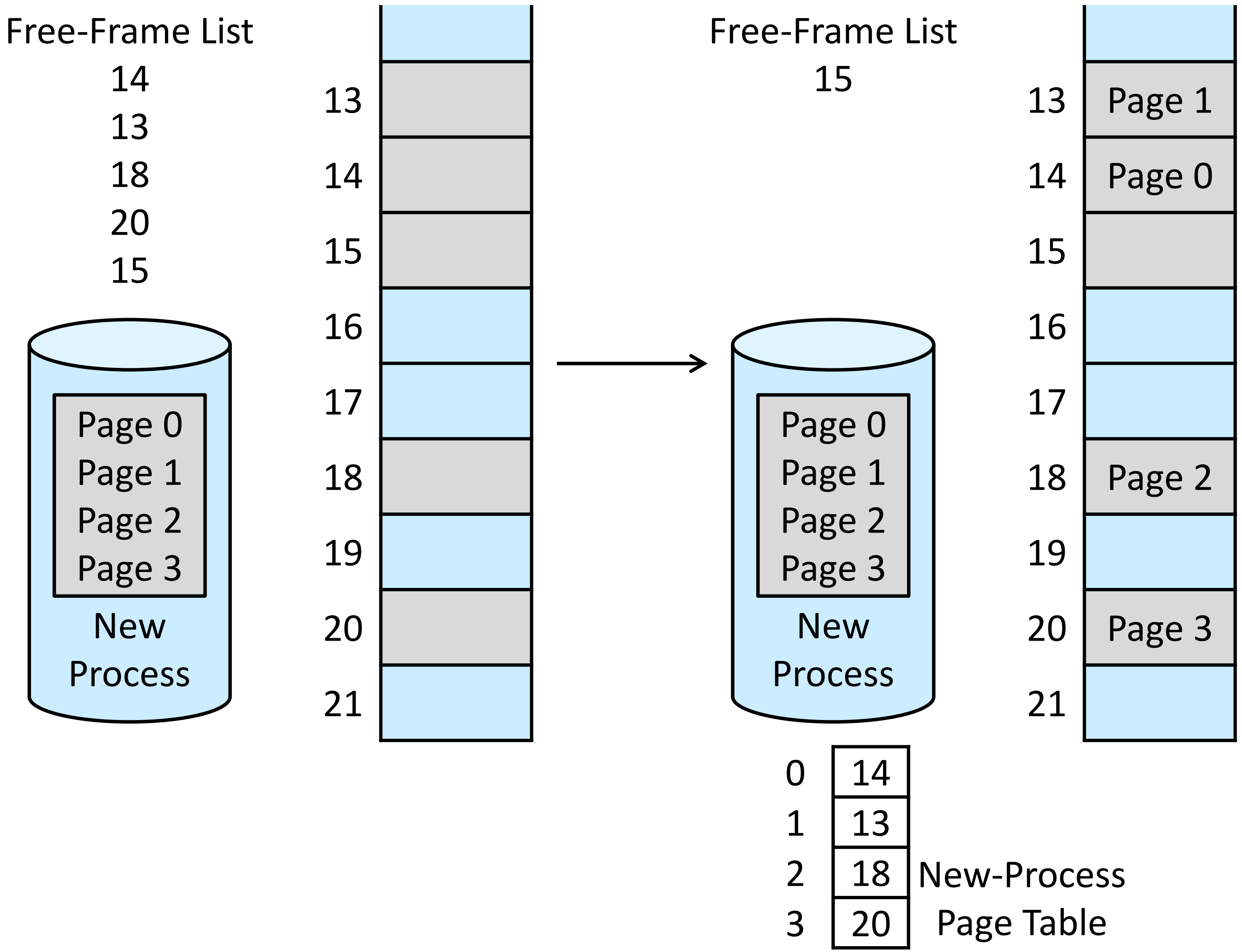


Paging Example

- m = 4
- n = 2
- Page size = 4 bytes
- # of frames = 8



Process Allocation



Paging's Internal Fragmentation

- Example
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes = 35 pages + 1,086 bytes
  - Internal fragmentation = 2,048 − 1,086 bytes = 962 bytes
- Worst-case fragmentation = 1 frame − 1 byte
- Average fragmentation = 0.5 frame
- Small page sizes are desirable?
  - The overhead of page table entries decreases as the page size increases
  - Disk I/O is more efficient when the amount of transferred data is larger
- Generally, page sizes have grown over time as processes, data sets, and main memory have become larger
  - Today, pages are typically either 4 KB or 8 KB in size
  - Some systems support even larger page sizes

Amount of Addressed Memory

- Frequently, on a 32-bit CPU, each page-table entry is 4-byte long, but that size can vary as well
  - A 4-byte entry can point to one of 2^{32} physical page frames
  - If the frame size is 4 KB (2^{12} bytes), then a system with 4-byte entries can address 16 TB (2^{44} bytes) of physical memory
- However, a system with 4-byte page-table entries may address less physical memory than the possible maximum
  - Other information that must be kept in the page-table entries
  - We will see in the later slides

Some Paging Facts

- An important aspect of paging
  - The clear separation between the programmer's view of memory and the actual physical memory
- **Frame table**
  - A single, **system-wide** data structure having one entry for each physical frame, indicating
    - Whether a frame is free or allocated
    - If it is allocated, to which page of which process (or processes)
- The operating system maintains a copy of the page table for each process

Outline

- Background
- Contiguous Memory Allocation
- **<u>Paging</u>**
  - Basic Method
  - **<u>Hardware Support</u>**
  - Protection
  - Shared Pages
- Structure of the Page Table
- Swapping
- Example: Intel 32- and 64-bit Architecture
- Example: ARMv8 Architecture

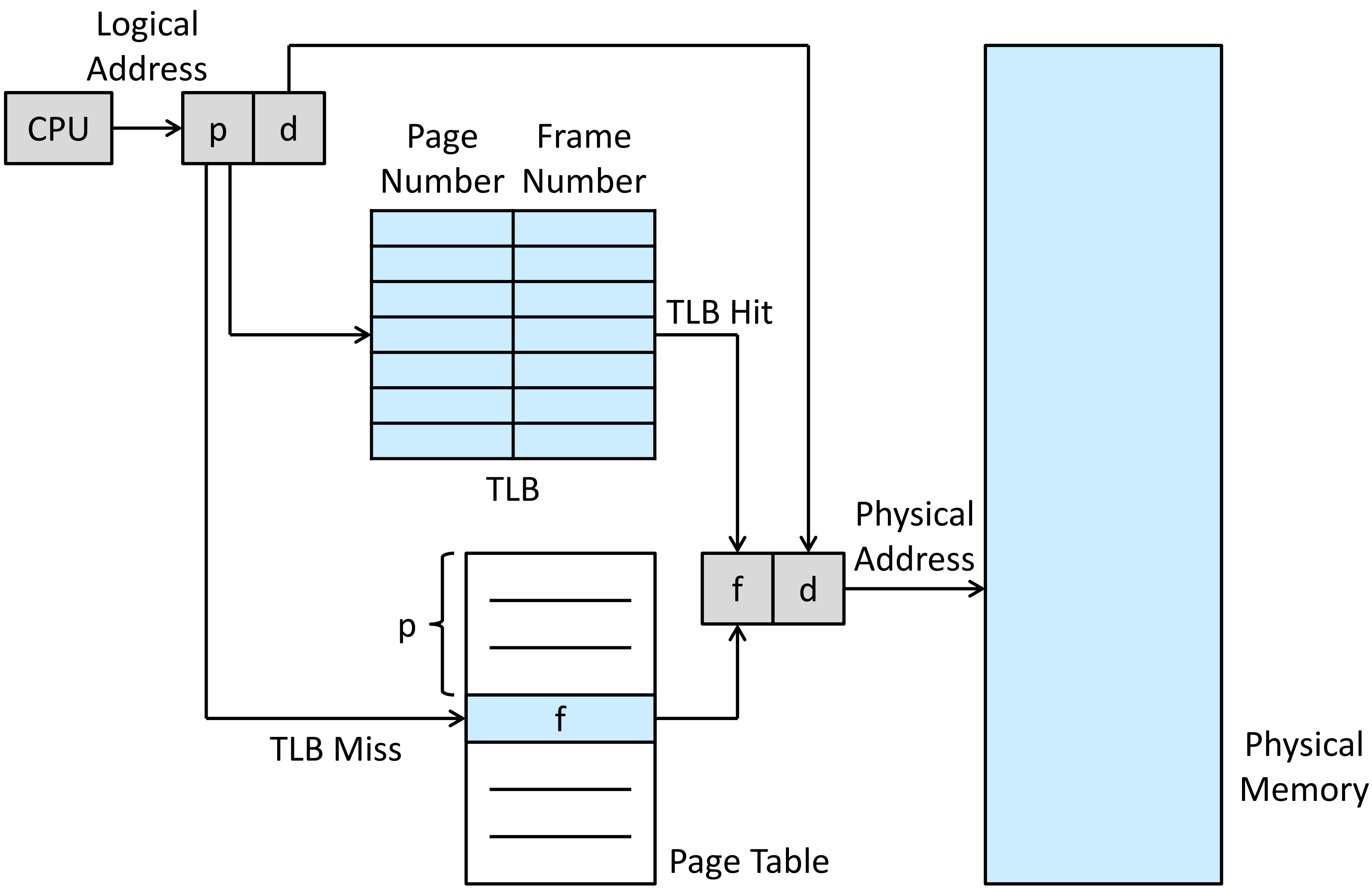
Page Table

- Page tables are per-process data structures
- Ways of implementation
  - A set of dedicated high-speed hardware registers
    - Make the page-address translation very efficient
    - Increase context-switch time, as each one of these registers must be exchanged during a context switch
  - A **page-table base register** (PTBR) stored in the process control block points to the page table stored in main memory
    - Change only the one register when changing page tables
    - Reduce context-switch time

Translation Look-Aside Buffer (1/3)

- Storing the page table in main memory yields faster context switches but results in slower memory access times
  - Two memory accesses are needed to access data (one for the page-table entry and one for the actual data)
- Standard solution: **translation look-aside buffer** (TLB), also called associative memory
  - A special, small, fast-lookup hardware cache
  - Each TLB entry consisting of two parts: a key (or tag) and a value
  - Typically number of TLB entries: 32 to 1,024 entries
- Systems have evolved from having no TLBs to having multiple levels of TLBs
  - Just as they have multiple levels of caches

Translation Look-Aside Buffer (2/3)



Translation Look-Aside Buffer (3/3)

- When a logical address is generated by the CPU, the MMU first checks if its page number is present in the TLB
  - If the page number is found, its frame number is used to access memory
    - Part of the instruction pipeline within the CPU, adding no performance penalty compared with a system that does not implement paging
  - If the page number is not in the TLB (known as a **TLB miss**), a memory reference to the page table must be made
    - Also add the page number and frame number to the TLB, so that they will be found quickly on the next reference
- If the TLB is full, an existing entry is selected for replacement
  - Examples: least recently used (LRU), round-robin, random, etc.
  - Some CPUs allow the OS to participate in LRU entry replacement
  - Some TLBs allow certain entries to be **wired down**
    - They (e.g., key kernel code) cannot be removed from the TLB

Address-Space Identifier

- Some TLBs store **address-space identifiers** (ASIDs) in each TLB entry
  - Identify each process
  - Provide address-space protection for that process
  - Allow the TLB to contain entries for several different processes simultaneously
- Without ASIDs, every time a new page table is selected (e.g., context switch), the TLB must be **flushed** (or erased)

Effective Memory-Access Time

- **Hit ratio**
  - Percentage of times that the page number of interest is found in the TLB
- Example: 10 nanoseconds to access memory
  - If the page number is in the TLB
    - A mapped-memory access takes 10 nanoseconds
  - If the page number is not in the TLB
    - Access memory for the page table and frame number (10 nanoseconds)
    - Access memory for the desired byte (10 nanoseconds)
    - A total of 20 nanoseconds
  - **Effective memory-access time**
    - 80-percent hit ratio: 0.8 * 10 + 0.2 * 20 = 12 (nanoseconds)
    - 99-percent hit ratio: 0.99 * 10 + 0.01 * 20 = 10.1 (nanoseconds)
- Calculation is more complicated with multiple levels of TLBs

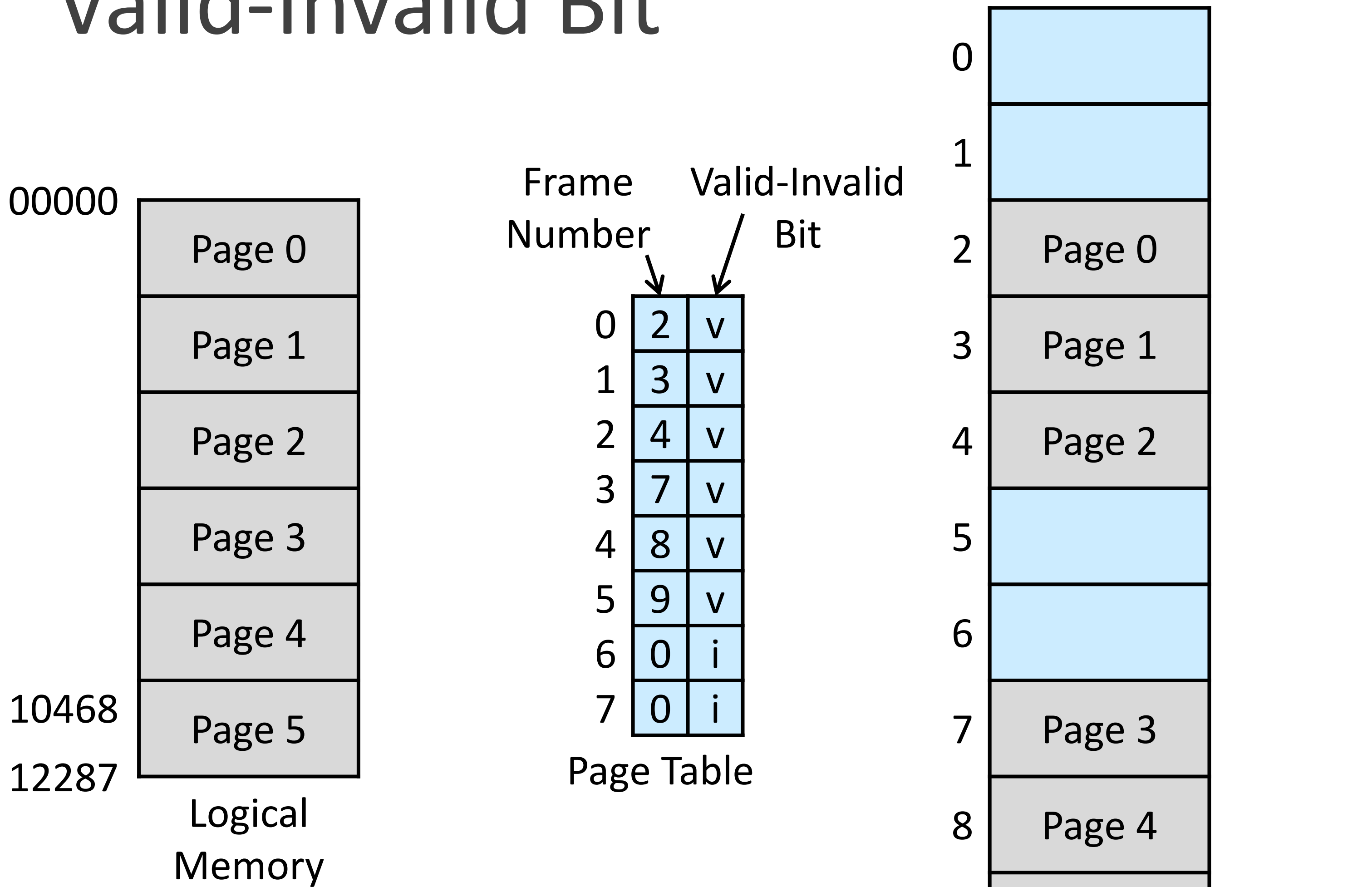
Outline

- ❑ Background
- ❑ Contiguous Memory Allocation
- ❑ **Paging**
  - ➢ Basic Method
  - ➢ Hardware Support
  - ➢ **Protection**
  - ➢ Shared Pages
- ❑ Structure of the Page Table
- ❑ Swapping
- ❑ Example: Intel 32- and 64-bit Architecture
- ❑ Example: ARMv8 Architecture

Protection

- Memory protection is accomplished by protection bits associated with each frame
  - Keep these bits in the page table
  - Provide read-only, read-write, execute-only, or combined protection
  - Trap illegal attempts to the operating system
- **Valid-invalid** bit
  - When this bit is set to valid, the associated page is in the process's logical address space
  - When the bit is set to invalid, the associated page is not in the process's logical address space
- **Page-table length register** (PTLR) to check the page-table size
  - Indicate the size of the page table
  - Check a logical address to verify that it is in the valid range

Valid-Invalid Bit



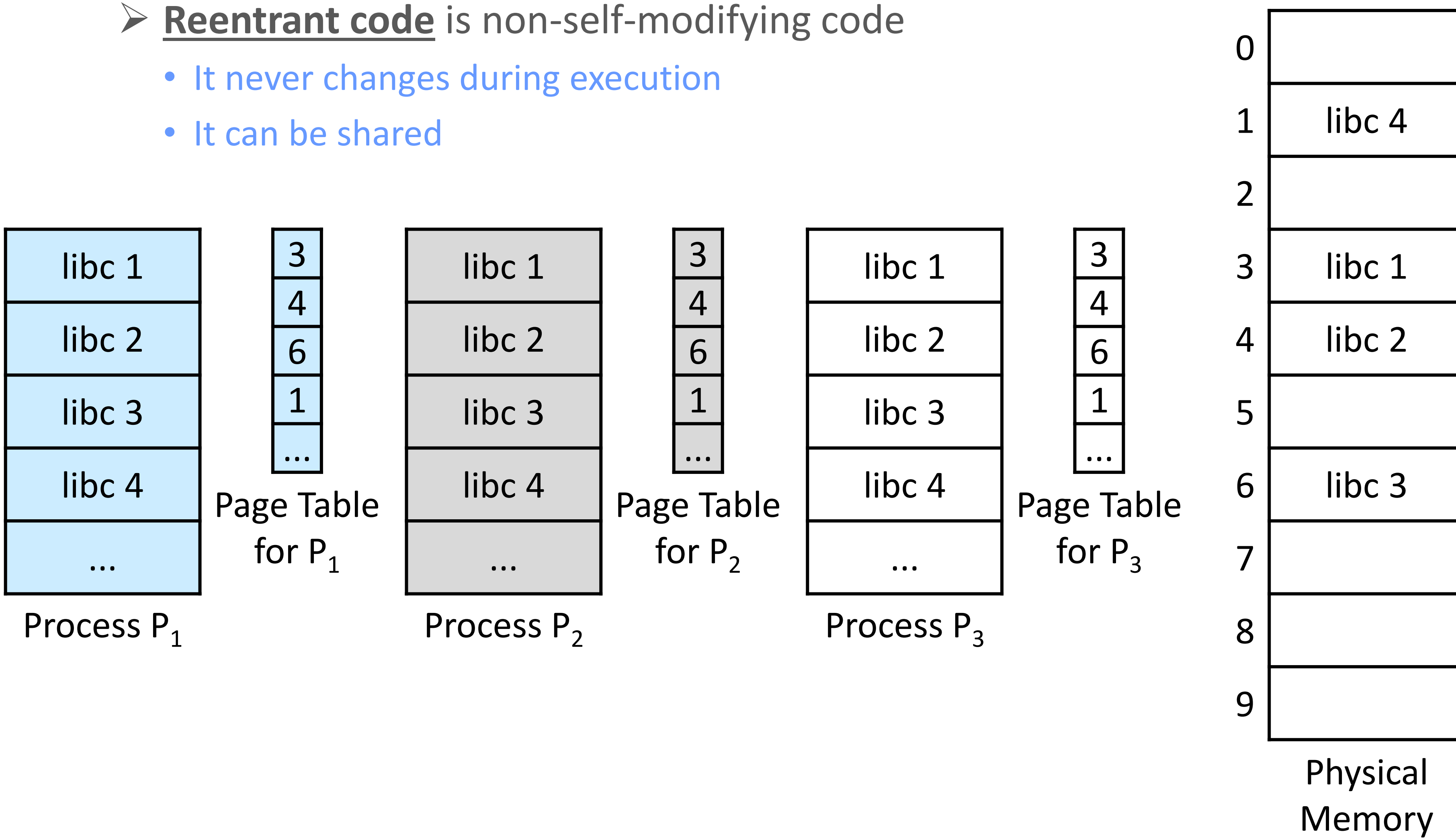
- What if a process uses only addresses 0 to 10468? A result of the 2-KB page size

Outline

- Background
- Contiguous Memory Allocation
- **Paging**
  - Basic Method
  - Hardware Support
  - Protection
  - **Shared Pages**
- Structure of the Page Table
- Swapping
- Example: Intel 32- and 64-bit Architecture
- Example: ARMv8 Architecture

Shared Pages

- A paging advantage: the possibility of sharing common code
  - **Reentrant code** is non-self-modifying code
    - It never changes during execution
    - It can be shared



Outline

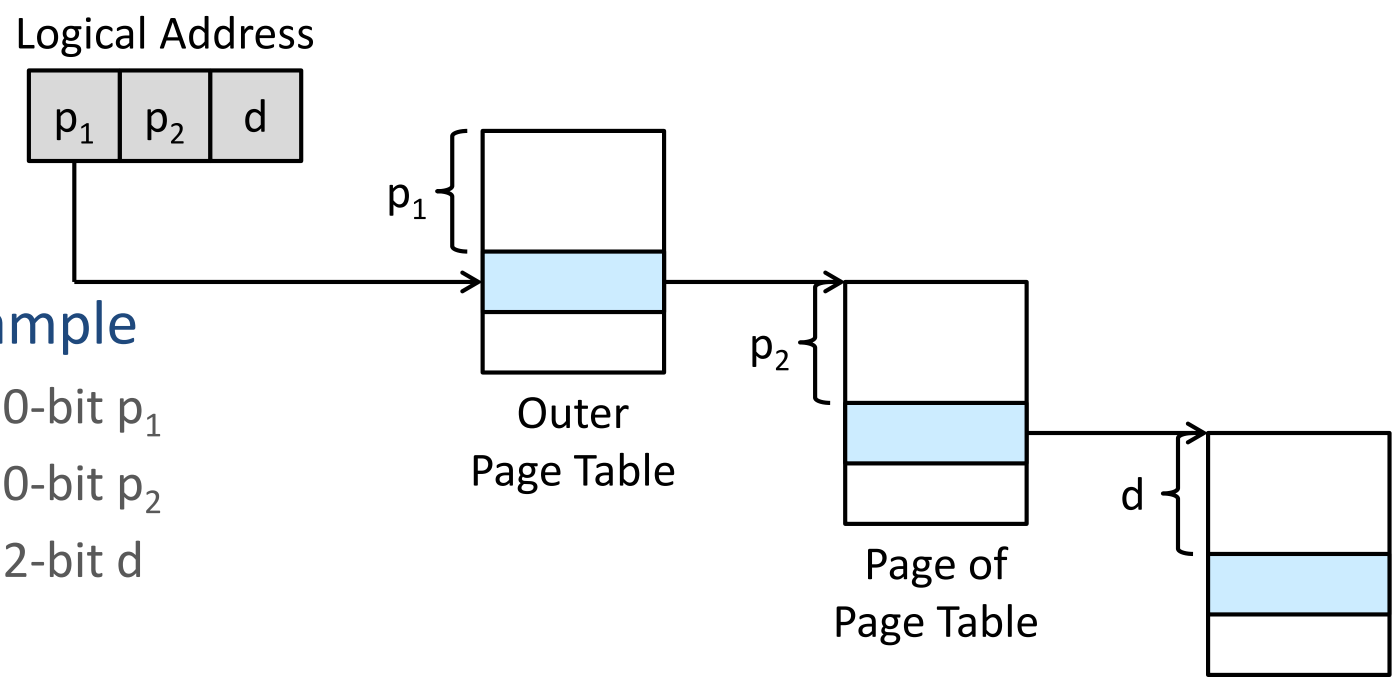
- Background
- Contiguous Memory Allocation
- Paging
- **<u>Structure of the Page Table</u>**
  - **<u>Hierarchical Paging</u>**
  - Hashed Page Tables
  - Inverted Page Tables
  - Oracle SPARC Solaris
- Swapping
- Example: Intel 32- and 64-bit Architecture
- Example: ARMv8 Architecture

Size of Page Table

- The page table itself may become excessively large
  - Most modern computer systems support a large logical address space (2^{32} to 2^{64} bytes)
  - If the page size 4 KB (2^{12} bytes), then a page table consists of 2^{20} entries
    - $2^{20} = 2^{32}/2^{12}$
  - If each entry consists of 4 bytes, then each process needs up to 4 MB of physical address space for the page table alone
    - We do not want to allocate the page table contiguously in main memory
    - One simple solution is to divide the page table into smaller pieces

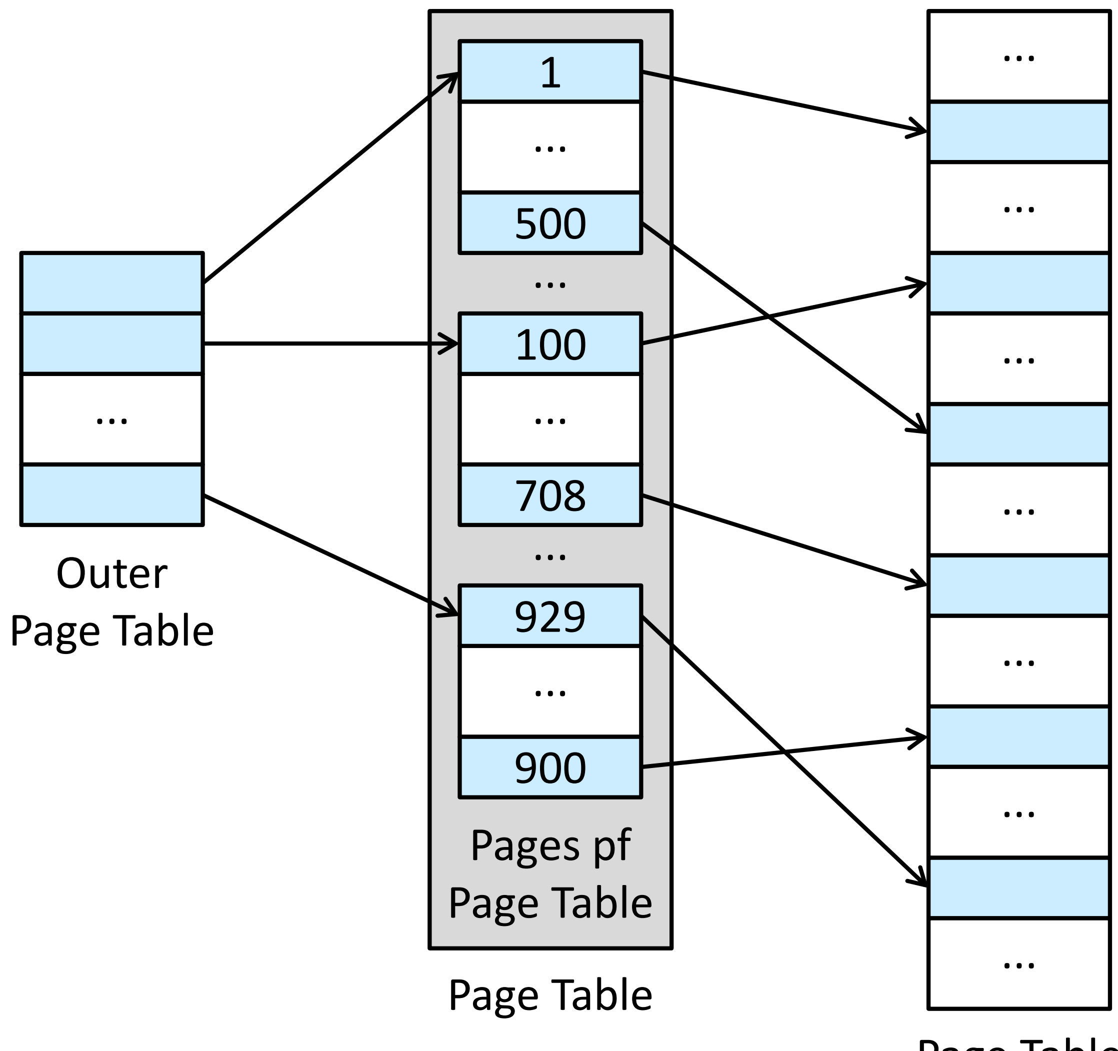
Two-Level Paging

- Divide the page number and page the page table
- **Forward-mapped page table**



- Example
  - 10-bit p_1
  - 10-bit p_2
  - 12-bit d

Two-Level Paging



Multi-Level Paging

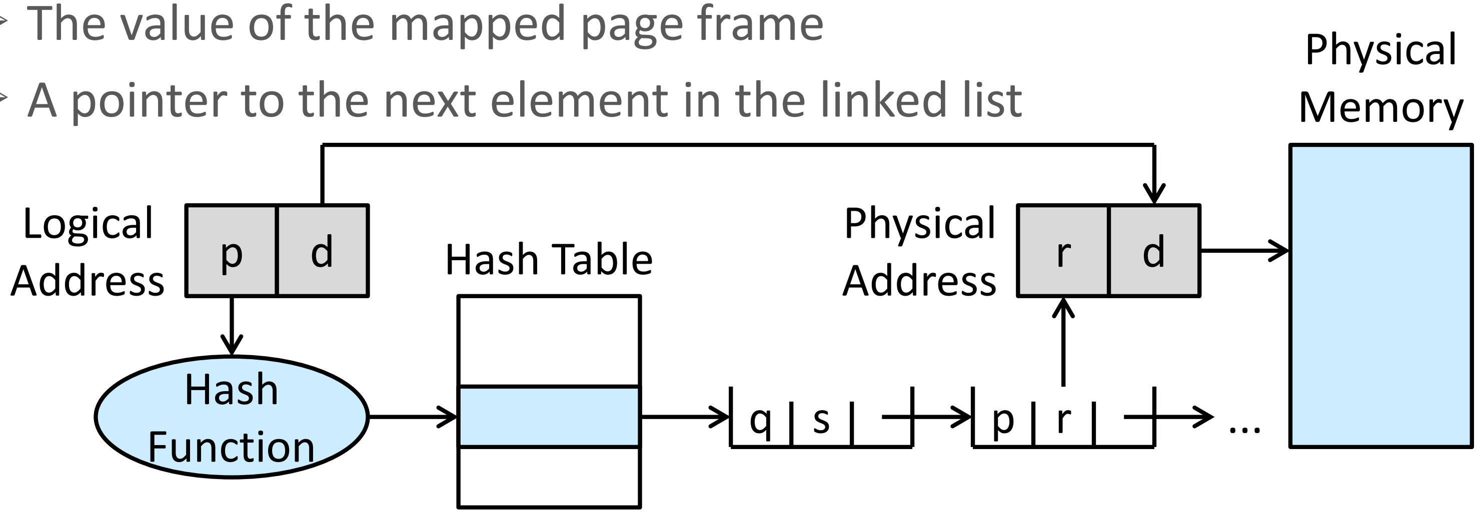
- For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate
- Example
  - If the page size 4 KB (2^{12} bytes), then a page table consists of 2^{52} entries
  - If we use a two-level paging scheme, then the inner page tables can conveniently be one page long, i.e., containing 2^{10} 4-byte entries
  - p_1, p_2, d = 42, 10, 12
  - p_1, p_2, p_3, d = 32, 10, 10, 12
  - ...
- The 64-bit UltraSPARC would require seven levels of paging
  - A prohibitive number of memory accesses

Outline

- Background
- Contiguous Memory Allocation
- Paging
- **Structure of the Page Table**
  - Hierarchical Paging
  - **Hashed Page Tables**
  - Inverted Page Tables
  - Oracle SPARC Solaris
- Swapping
- Example: Intel 32- and 64-bit Architecture
- Example: ARMv8 Architecture

Hashed Page Tables

- Each entry in the hash table contains a linked list of elements
- Each element consists of three fields
  - The virtual page number
  - The value of the mapped page frame
  - A pointer to the next element in the linked list



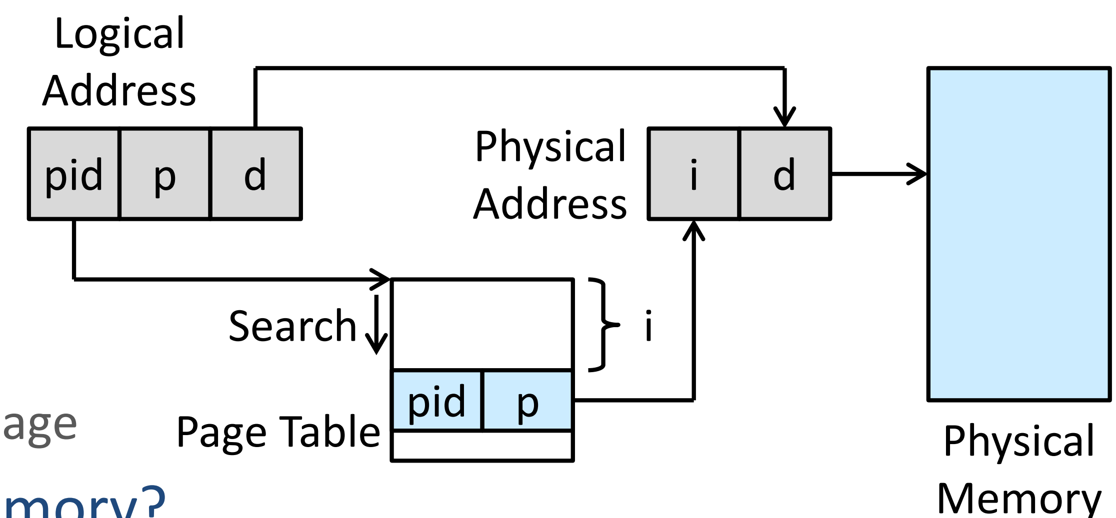
- Variation: **clustered page tables**
  - Similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page

Outline

- Background
- Contiguous Memory Allocation
- Paging
- **Structure of the Page Table**
  - Hierarchical Paging
  - Hashed Page Tables
  - **Inverted Page Tables**
  - Oracle SPARC Solaris
- Swapping
- Example: Intel 32- and 64-bit Architecture
- Example: ARMv8 Architecture

Inverted Page Tables

- One entry for each real page (or frame) of memory
  - Decrease the amount of memory needed to store each page table
  - Increase the amount of time needed to search the table
    - Use a hash table to limit the search to one or a few page-table entries
    - Use a TLB to accelerate
- Each entry consists of
  - The virtual address of the page stored in that real memory location
  - Information about the process that owns the page



- How about shared memory?
  - Only one mapping of a virtual address to the shared physical address may occur at any given time

Outline

- ❑ Background
- ❑ Contiguous Memory Allocation
- ❑ Paging
- ❑ **Structure of the Page Table**
  - ➢ Hierarchical Paging
  - ➢ Hashed Page Tables
  - ➢ Inverted Page Tables
  - ➢ **Oracle SPARC Solaris**
- ❑ Swapping
- ❑ Example: Intel 32- and 64-bit Architecture
- ❑ Example: ARMv8 Architecture

Oracle SPARC Solaris

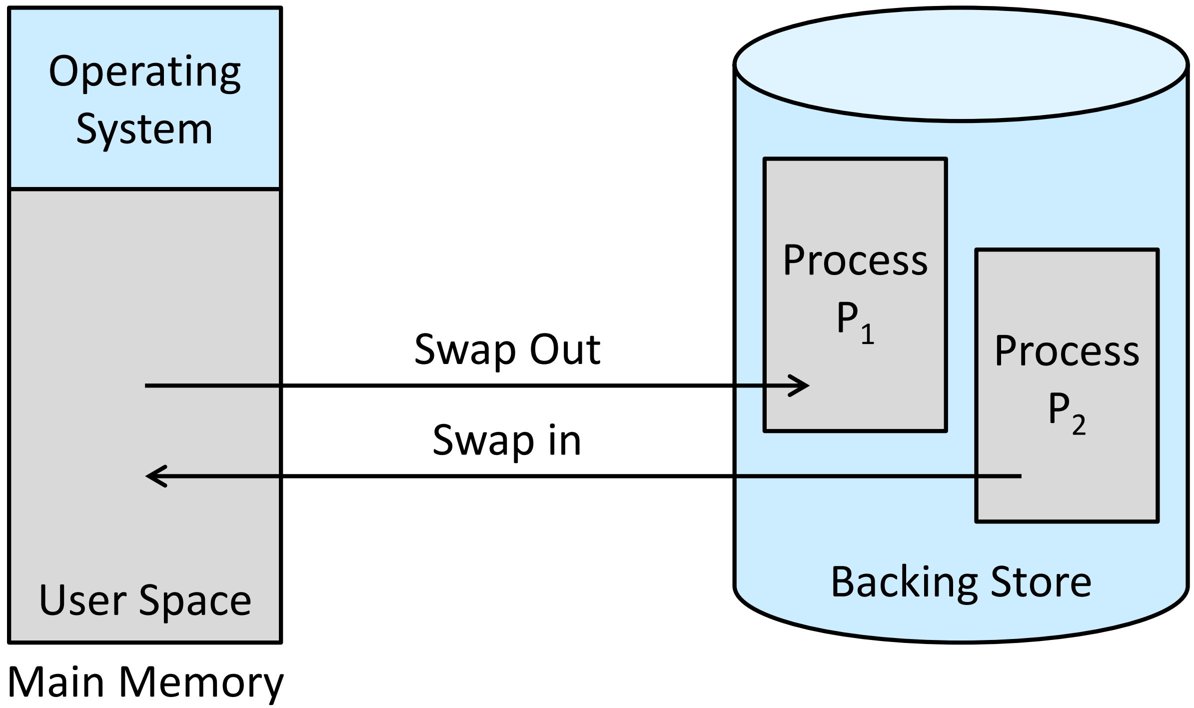
- **Solaris** on the **SPARC** CPU is a fully 64-bit operating system
  - Not use all of its physical memory by keeping multi-level page tables
- Two hashed page tables
  - One for the kernel and one for all user processes
  - Each table maps memory addresses from virtual to physical memory
  - Each entry represents a contiguous area of mapped virtual memory
    - More efficient than having a separate hash-table entry for each page
  - Each entry has a base address and a span indicating the number of pages the entry represents
- Searching through a hash table would take too long
  - A TLB holds translation table entries (TTEs)
  - A cache of these TTEs resides in a translation storage buffer (TSB)
  - **TLB walk**: search TLB and, if needed, TSB for a TTE

Outline

- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- **Swapping**
  - Standard Swapping
  - Swapping with Paging
  - Swapping on Mobile Systems
- Example: Intel 32- and 64-bit Architecture
- Example: ARMv8 Architecture

Swapping

- A process, or a portion of a process, can be
  - **Swapped** temporarily out of memory to a **backing store**
  - Brought back into memory for continued execution
- Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory

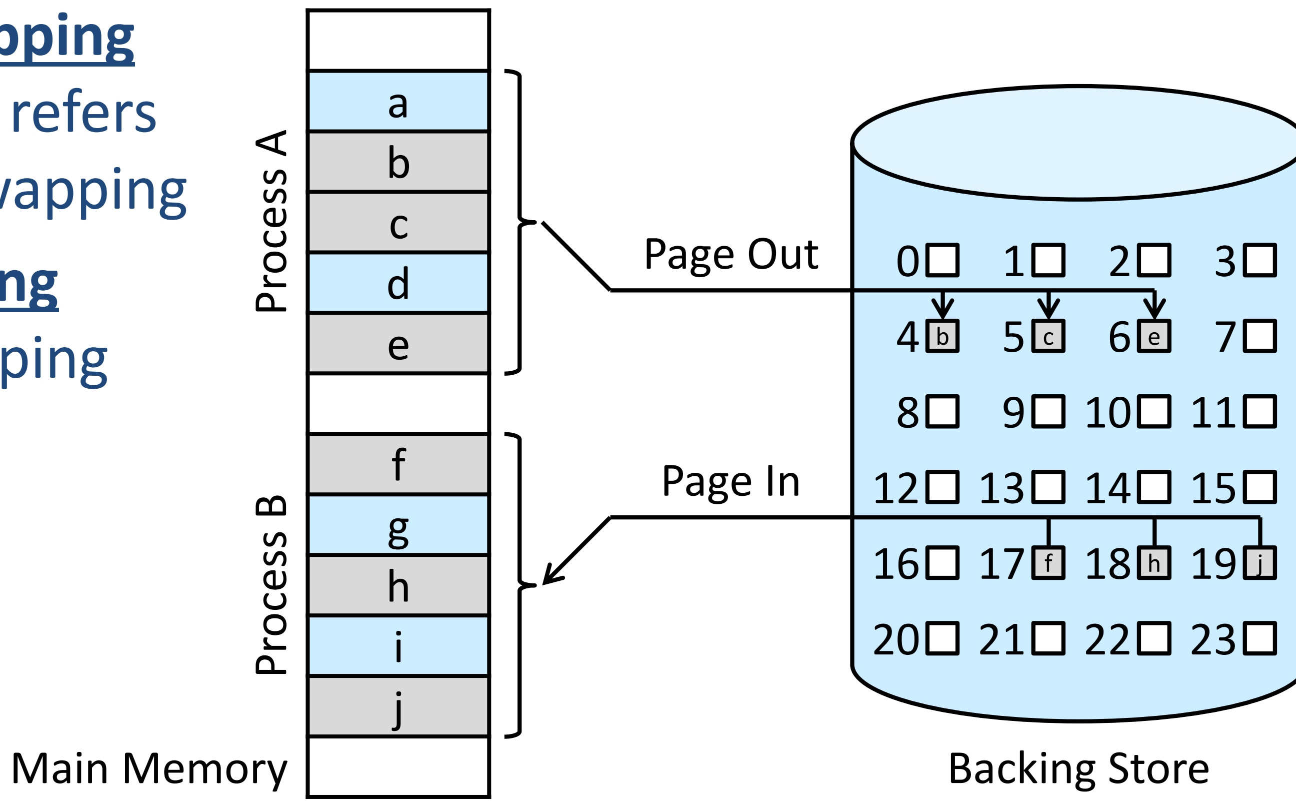


Standard Swapping

- Standard swapping involves moving **<u>entire</u>** processes between main memory and a backing store
  - The backing store is commonly fast secondary storage
  - When a process is swapped out, its data structures must be written to the backing store
    - For a multithreaded process, all per-thread data structures must be swapped as well
- Standard swapping allows physical memory to be oversubscribed
  - Idle or mostly idle processes are good candidates for swapping
  - If a process that has been swapped out becomes active, it must then be swapped back in

Swapping with Paging

- Most systems now (Linux and Windows) use a variation of swapping in which pages of a process can be swapped
  - Still allows physical memory to be oversubscribed
  - Does not incur the cost of swapping entire processes
- The term **swapping** now generally refers to standard swapping
- The term **paging** refers to swapping with paging
- Chapter 10



Swapping on Mobile Systems

- Mobile systems typically do not support swapping in any form
  - Generally use flash memory rather than more spacious hard disks for nonvolatile storage
    - Small amount of space
    - Limited # of writes that flash memory can tolerate before becoming unreliable
    - Poor throughput between main memory and flash memory
- When free memory falls below a certain threshold
  - iOS asks applications to voluntarily relinquish allocated memory
    - Read-only data are removed from main memory
    - Data that have been modified are never removed
    - Any application that fails to free up sufficient memory may be terminated
  - Android adopts a strategy similar to that used by iOS
    - Before terminating a process, Android writes its **application state** to flash memory

Outline

- ❑ Background
- ❑ Contiguous Memory Allocation
- ❑ Paging
- ❑ Structure of the Page Table
- ❑ Swapping
- ❑ **Example: Intel 32- and 64-bit Architecture**
- ❑ Example: ARMv8 Architecture

Intel 32- and 64-bit Architecture

- History
  - 16-bit chips Intel 8086 and Intel 8088
  - A series of 32-bit chips, IA-32, included the family of 32-bit Pentium processors
  - A series of 64-bit chips based on x86-64 architecture
- Most current/popular PC operating systems run on Intel chips
  - However, the dominance has not spread to mobile systems, where the ARM architecture currently enjoys considerable success

IA-32 Architecture

- The segmentation and paging units form the equivalent of the memory-management unit (MMU)
  - The CPU generates logical addresses, which are given to the segmentation unit
  - The segmentation unit produces a linear address for each logical address, which is given to the paging unit
  - The paging unit generates the physical address in main memory
- IA-32 segmentation
  - The logical address space of a process is divided into two partitions
    - The first partition consists of segments that are private to that process
    - The second partition consists of segments that are shared among all processes
- IA-32 paging
  - A two-level paging scheme where p_1, p_2, d = 10, 10, 12

x86-64

- Intel adopted AMD's x86-64 architecture
  - Historically, AMD had often developed chips based on Intel's architecture
- Support for a 64-bit address space yields 2^{64} bytes of addressable memory
- The x86-64 architecture currently provides a 48-bit virtual address
  - Support for 52-bit physical addresses with **page address extension**
  - Support for page sizes of 4 KB, 2 MB, or 1 GB
  - Four levels of paging hierarchy

Outline

- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Example: Intel 32- and 64-bit Architecture
- **Example: ARMv8 Architecture**

ARMv8 Architecture

- In addition to mobile devices, ARM also provides architecture designs for real-time embedded systems
- The ARMv8 has three different **translation granules**
  - Each translation granule provides different page sizes, as well as larger sections of contiguous memory, known as **regions**

Translation Granule Size	Page Size	Region Size	Paging
4 KB	4 KB	2 MB, 1 GB	Up to 4 Levels
16 KB	16 KB	32 MB	Up to 4 Levels
64 KB	64 KB	512 MB	Up to 3 Levels

- The ARM architecture also supports two levels of TLBs
  - At the inner level are two **micro TLBs**
    - One for data and one for instructions; ASIDs supported
  - At the outer level is a single **main TLB**
  - Address translation begins at the micro-TLB level

Objectives

- Explain the difference between a logical and a physical address and the role of the memory management unit (MMU) in translating addresses
- Apply first-, best-, and worst-fit strategies for allocating memory contiguously
- Explain the distinction between internal and external fragmentation
- Translate logical to physical addresses in a paging system that includes a translation look-aside buffer (TLB)
- Describe hierarchical paging, hashed paging, and inverted page tables

Q&A

Exercises

- 9.4
  - Consider a logical address space of 64 pages of 1,024 words each, mapped onto a physical memory of 32 frames
    - How many bits are there in the logical address?
    - How many bits are there in the physical address?
- 9.10
  - Consider a computer system with a 32-bit logical address and 4-KB page size
  - The system supports up to 512 MB of physical memory
  - How many entries are there in each of the following?
    - A conventional, single-level page table
    - An inverted page table